МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки

ХОМИШИН Віктор Григорович

Метод та засоби захисту ASP.NET Core веб-додатків / Method and Tools for ASP.NET Core Securing in Web Applications

спеціальність: 125 – Кібербезпека та захист інформації

освітньо-професійна програма – Кібербезпека

Кваліфікаційна робота

Виконав студент групи
КБзм -21
В.Г. Хомишин

Науковий керівник
к.т.н., доцент Т.Г. Цаволик

Кваліфікаційну роботу
допущено до захисту:

«____» _____ ___2024 р.

Завідувач кафедри
_____ **В.В. Яцків**

ТЕРНОПІЛЬ - 2024

Факультет комп'ютерних інформаційних технологій
Кафедра кібербезпеки
Освітній ступінь «магістр»
спеціальність: 125 – Кібербезпека та захист інформації
освітньо-професійна програма – Кібербезпека

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ В.В. Яцків
«____» _____ 20___ року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

ХОМИШИН ВІКТОР ГРИГОРОВИЧ
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Метод та засоби захисту ASP.NET Core веб-додатків / Method and Tools for ASP.NET Core Securing in Web Applications
керівник роботи: к.т.н., доцент Т.Г. Цаволик
затверджені наказом по університету від 12 грудня 2023 року № 753

2. Строк подання студентом закінченої кваліфікаційної роботи 12 грудня 2024 р.

3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити:
– описати можливості та особливості фреймворку ASP.NET Core;
– розглянути типи створюваних додатків;
– описати компоненти та загрози типового веб-додатку;
– запропонувати захист від міжсайтового скриптингу;
– запропонувати захист від міжсайтової підробки запиту;
– розглянути безпеку використання ресурсів з різних джерел;
– розробити алгоритм запобігання атакам з відкритим перенаправленням;
– описати додавання глобальної політики CORS до всього додатку;
– розробити захист від атак з використанням впровадження SQL-коду;
– розширити базові можливості ASP.NET Core Identity;

5. Перелік графічного матеріалу у роботі:
– структура типового додатку ASP.NET Core, типи додатків та взаємодій;
– робочі вікна середовища розробки та описаних веб-додатків;
– лістинг коду реалізації пропонованих заходів захисту веб-додатків.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання: 12 грудня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строки виконання етапів кваліфікаційної роботи	Примітка
1	Аналіз предметної області	12.2023 р. – 03.2024 р.	
2	Методи захисту ASP.NET Core веб-додатків	03.2024 р. – 06.2024 р.	
3	Удосконалення засобів захисту веб-додатків	06.2024 р. – 11.2024 р.	

Студент _____ В.Г. Хомишин

Керівник роботи _____ к.т.н., доц. Т.Г. Цаволик

АНОТАЦІЯ

Кваліфікаційна робота на тему «Метод та засоби захисту ASP.NET Core веб-додатків» на здобуття освітнього ступеня «Магістр» зі спеціальності 125 «Кібербезпека та захист інформації» освітньо-професійної програми «Кібербезпека» написана обсягом 98 сторінок і містить 53 ілюстрації, 3 таблиці, 1 додаток та 34 джерела за переліком посилань.

Метою роботи є підвищення ефективності захисту веб-додатків, розроблених на базі фреймворку ASP.NET Core.

Методи досліджень. Для розв'язання поставлених задач у даній кваліфікаційній роботі використано: методи аналізу архітектури, кіберзагроз, тестування безпеки, алгоритми аналізу запитів та коду додатків, методи проектування.

Результати дослідження: запропоновано механізми захисту від міжсайтового скриптингу та міжсайтової підробки запиту; розроблено алгоритм запобігання атакам з відкритим перенаправленням; запропоновано реалізацію глобальної політики CORS до всього додатку; удосконалено захист від атак з використанням впровадження SQL-коду; розширено базові можливості системи автентифікації та авторизації ASP.NET Core Identity.

Результати роботи можуть бути застосовані при розробці та впровадженні безпечних веб-додатків на основі платформи ASP.NET Core, підвищенні рівня захищеності корпоративних систем та оптимізації процесів аудиту та тестування безпеки веб-додатків на базі зазначеного фреймворку.

Ключові слова: ВЕБ-ДОДАТОК, ASP.NET CORE, ВРАЗЛИВОСТІ, ЗАГРОЗИ, ЗАХИСТ ІНФОРМАЦІЇ, АУДИТ БЕЗПЕКИ.

ABSTRACT

Qualification work on "Method and Tools for ASP.NET Core Securing in Web Applications" for the degree of "Master" in the specialty 125 "Cybersecurity and Information Protection" educational and professional program "Cybersecurity" is written in 98 pages and contains 53 illustrations, 3 tables, 1 appendix and 34 source according to the list of links.

The purpose of the work is to improve the protection efficiency of web applications developed based on the ASP.NET Core framework.

Research methods. To solve the tasks in this qualification work, the following methods were used: methods of architecture analysis, cyber threats, security testing, algorithms for analysing requests and application code, design methods.

Research results: proposes mechanisms for protecting against cross-site scripting and cross-site request forgery; develops an algorithm for preventing open redirect attacks; proposes the implementation of a global CORS policy for the entire application; improves protection against attacks using SQL injection; expands the basic capabilities of the ASP.NET Core Identity authentication and authorization system.

The results of the work can be applied in the development and implementation of secure web applications based on the ASP.NET Core platform, increasing the level of security of corporate systems, and optimizing the processes of auditing and testing the security of web applications based on the specified framework.

Keywords: WEB APPLICATION, ASP.NET CORE, VULNERABILITIES, THREATS, INFORMATION PROTECTION, SECURITY AUDIT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API	–	Application Programming Interface
ASP	–	Active Server Pages
CORS	–	Cross-Origin Resource Sharing
CSRF	–	Cross-Site Request Forgery
JSON	–	JavaScript Object Notation
HTTP	–	Hypertext Transfer Protocol
MVC	–	Model-View-Controller
SQL	–	Structured Query Language
TOTP	–	Time-based One-time Password algorithm
XML	–	EXtensible Markup Language
XSS	–	Cross Site Scripting

ВСТУП

Актуальність роботи. Не проходить й тижня без різноманітних гучних інцидентів у галузі інформаційної безпеки. Це може бути витік даних у відкритий доступ, поновлення популярних бібліотек зі шкідливим кодом, поширення нових програм-вимагачів, підозрілі й небезпечні сайти, піддані вразливостям. Багато з подій, про які ми дізнаємося з IT-новин, стали можливими завдяки помилкам у коді.

Загальна безпека веб-додатків в останні роки продовжує покращуватися, але все ще залишає бажати кращого [1, 2]. Дослідження безпекових лабораторій, проведені у 2020 році, дають наступні результати [3]:

– хакери можуть атакувати користувачів у 9 із 10 веб-додатків. Атаки включають перенаправлення користувачів на ресурс, контрольований хакерами, викрадення облікових даних під час фішингових атак та зараження комп'ютерів шкідливим програмним забезпеченням;

– несанкціонований доступ до додатків можливий на 39 % сайтів. У 2019 році повний контроль над системою можна було отримати над 16 % веб-додатків. У 8 % систем повний контроль над сервером веб-додатків дозволяв атакувати локальну мережу;

– порушення конфіденційних даних були загрозою для 68 % веб-додатків. Більшість зламаних даних стосувалися особистого характеру (47 % порушень) або облікових даних (31 %).

Щодо статистики вразливостей, дані теж не втішні:

– 82 % вразливостей знаходилися в коді програми;

– кожна досліджена система містила в середньому 22 вразливості, з яких 4 були високої серйозності;

– кожна п'ята вразливість має високі ризики.

Ґрунтуючись на цьому, можна виділити дві тенденції:

– основна загроза безпеки веб-додатків полягає в коді;

– проблема носить масштабний характер, тому цим питанням варто

приділяти більше уваги.

Дуже часто недоліки в системі безпеки проявляються не одразу, а лише тоді, коли стає надто пізно й веб-додаток успішно зламаний. Тому необхідно зробити безпеку веб-додатків головним пріоритетом й використовувати передові методи для її забезпечення із самого початку проекту [4].

Більшість загроз пов'язані з роботою веб-браузерів, протоколу HTTP, серверів баз даних та іншими мережевими аспектами [5]. Отже, ці ризики не залежать від технологій [33]. От один із прикладів – це впровадження коду JavaScript на сайт, працює незалежно від використовуваного мови сервера або фреймворку. На практиці існують наступні відмінності:

1. Деякі мови й фреймворки мають вбудовані засоби протидії, які допомагають запобігти розповсюдженим атакам без яких-небудь додаткових зусиль на етапі розробки.

2. У різних технологіях та фреймворках функції, методи й API, що використовуються для захисту від певних атак і ризиків, називаються по-різному.

Активний розвиток в останні роки фреймворку ASP.NET Core потребує забезпечення високого рівня безпеки веб-додатків шляхом інтеграції сучасних методів автентифікації, авторизації, захисту від атак типу SQL-ін’єкція, XSS, CSRF, а також безпечного управління сесіями користувачів.

Мета і завдання дослідження. Метою роботи є підвищення ефективності захисту веб-додатків, розроблених на базі фреймворку ASP.NET Core.

Досягнення визначеної мети передбачає вирішення таких завдань:

– вивчити можливості та особливості фреймворку ASP.NET Core;

– дослідити загрози створюваних веб-додатків;

– запропонувати заходи протидії для найбільш поширених типів загроз;

– дослідити можливість використання глобальних політик до додатків;

– розробити захист від атак з використанням впровадження SQL-коду;

– розширити базові можливості ASP.NET Core Identity.

Об’єкт дослідження – вразливості веб-додатків.

Предмет дослідження – процеси, властивості, характеристики та код додатків, що впливають на функціональність, безпеку, продуктивність та зручність використання веб-застосунків на базі фреймворку ASP.NET Core.

Методи досліджень. Для розв’язання поставлених у даній кваліфікаційній роботі задач використано методи аналізу архітектури, кіберзагроз, тестування безпеки, алгоритми аналізу запитів та коду додатків, методи проектування.

Наукова новизна одержаних результатів. Комплексно досліджено моделі загроз веб-застосунків на базі фреймворку ASP.NET Core та розроблено рекомендації щодо забезпечення захисту від поширених видів атак.

Практичне значення отриманих результатів. Розширено базові можливості компоненти ASP.NET Core Identity, що дозволяє використовувати всі параметри керування користувачами та їх входом, двофакторну автентифікацію з можливістю генерування додатком QR-кодів та можливістю реєстрування й проходження автентифікації шляхом використання сторонніх облікових записів.

Публікації та апробація КР.

1. Деркач М.В., Хомишин В.Г., Гудзенко В.О. Тестування безпеки вебресурсу на базі інструментів для сканування та виявлення вразливостей. Наукові вісті Далівського університету. Електронне видання. 2023. № 25. DOI: https://doi.org/10.33216/2222-3428-2023-25-1.

2. Хомишин В. Г. Захист ASP.NET Core веб-додатків від впровадження SQL-коду. Актуальні задачі сучасних технологій : Матеріали XIII Міжнар. науково-техн. конф. молодих уч. та студентів, м. Тернопіль, 11–12 груд. 2024 р. Тернопіль, 2024.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Загальна інформація про фреймворк ASP.NET Core

ASP.NET Core – це кросплатформний фреймворк з відкритим вихідним кодом, який можна використовувати для швидкого створення динамічних веб-додатків. Його можна використовувати для створення веб-додатків з формуванням на стороні сервера, серверних програм, API для протоколу HTTP, які можуть застосовуватися мобільними програмами, і багато іншого. ASP.NET Core працює на .NET 9, це його остання версія [6].

ASP.NET Core надає структуру, допоміжні функції та фреймворк для створення додатків, що позбавляє розробника необхідності писати більшу частину цього коду самостійно. Згодом код фреймворку ASP.NET Core викликає «обробники», які, своєю чергою, викликають методи бізнес-логіки програми, як показано на рисунку 1.1 [7]. Ця бізнес-логіка є ядром розроблюваної програми. Тут додаток може взаємодіяти з іншими службами, такими як бази даних або віддалені API, але бізнес-логіка зазвичай не залежить безпосередньо від ASP.NET Core [8].

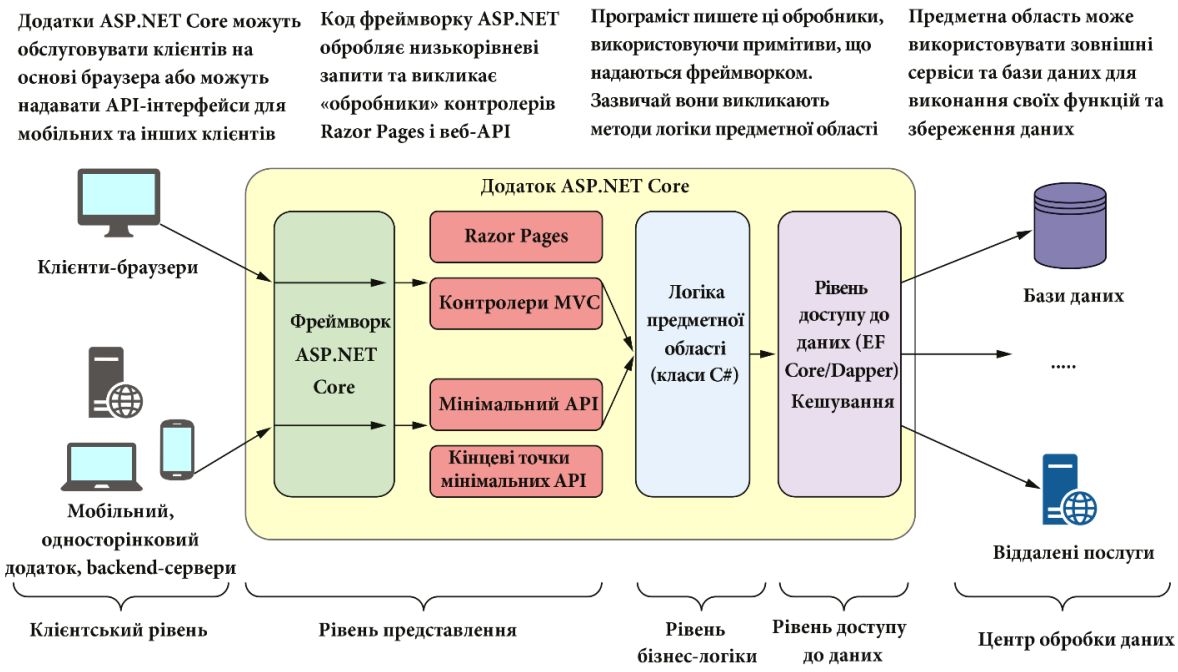


Рисунок 1.1 – Структура типового додатку ASP.NET Core

Типовий додаток ASP.NET Core складається із кількох рівнів [9-11]. Код фреймворку ASP.NET Core обробляє запити від клієнта, працюючи зі складним мережевим кодом. Згодом фреймворк викликає обробники (Razor Pages та контролери веб-API), які пишуться з використанням примітивів, що надаються фреймворком. Наприкінці ці обробники викликають логіку предметної області додатку, яка зазвичай є класами і об'єктами C# без будь-яких залежностей, специфічних для ASP.NET [12].

ASP.NET має довгу історію. Microsoft вперше випустила бета-версію фреймворку у 2001 р., а в якості остаточної версії 1.0 – на початку 2002 р. Тоді програмний пакет називався NET Framework й містив фреймворк для розробки серверних веб-додатків за назвою ASP.NET (перші три букви були взяті з назви попередньої веб-технології Microsoft ASP – скорочення від «Active Server Pages»). Разом с.NET Framework з'явилася й нова мова програмування C#. З того часу вона пройшла кілька випусків, у кожному з яких були додані нові функції та розширюваність. Однак кожен випуск був побудований на основі .NET Framework, тому вона встановлена на всіх версіях Windows.

1.2 Історія версій ASP.NET Core

ASP.NET та .NET Framework розвивалися протягом багатьох років. У 2010-х роках компанія Microsoft почала працювала над новою версією .NET, кульмінацією якої став випуск .NET Core 1.0 в середині 2016 р. Це була нова версія .NET з відкритим вихідним кодом, більш-менш незалежна від платформи й вона більше не була прив'язана до Windows. Слово Core використовувалося, щоб уникнути плутанини з .NET Framework, особливо що стосувалося номерів версій.

Остання й, імовірно, остаточна версія .NET Framework і ASP.NET – 4.8.1, вийшла 9 серпня 2022 року. З одного боку на сьогодні ASP.NET 4.x є надійною, перевіреною в роботі платформою для створення сучасних програм для ОС Windows. З іншого – вона обмежена цією залежністю: зміни в базовій

платформі .NET Framework мають далекосяжні наслідки, внаслідок чого спостерігається уповільнення швидкості розповсюдження, а це залишає за бортом багатьох розробників, які створюють та розгортають програми для Linux чи macOS.

Перший випуск фреймворку .NET Core у червні 2016 року мав багато зауважень, зокрема, що стосувалося роботи з інструментами (таблиця 1.1). З випуском .NET 7 у листопаді 2022 року ASP.NET Core дійсно став самостійним: API та інструменти досягли зрілого рівня. Оновлення фреймворку в .NET 6 та .NET 7, у яких завдяки введенню мінімального хостингу та мінімального API, забезпечується більш короткий і простий підхід до написання API, що набагато ближче до інших мов й значно спрощує процес початку роботи для новачків. Розробник можете одразу приступати до створення функціональності свого додатку без необхідності спочатку розбиратися в архітектурі.

Таблиця 1.1 – Історія версій фреймворку .NET Core

Версія	Дата випуску	Середовище розробки	Дата закінчення підтримки
.NET Core 1.0	27 червня 2016 р.	Visual Studio 2015 оновлення 3	27 червня 2019 р
.NET Core 2.0	14 серпня 2017 р.	Visual Studio 2017 версія 15.3	1 жовтня 2018 р.
.NET Core 3.0	23 вересня 2019 р.	Visual Studio 2019 версія 16.3	3 березня 2020 р.
.NET 5	10 листопада 2020 р.	Visual Studio 2019 версія 16.8	10 травня 2022 р.
.NET 6	8 листопада 2021 р.	Visual Studio 2022 версія 17.0	12 листопада 2024 р.
.NET 7	8 листопада 2022 р.	Visual Studio 2022 версія 17.4	14 травня 2024 р.
.NET 8	14 листопада 2023 р.	Visual Studio 2022 версія 17.8	10 листопада 2026 р.
.NET 9	12 листопада 2024 р.	Visual Studio 2022 версія 17.12	Травень 2026 р.

Середовище.NET 8 включає покращення продуктивності, збирання сміття та удосконалення основних бібліотек розширень. Воно також має новий режим глобалізації для мобільних додатків й нових генераторів джерел для COM взаємодії та прив'язки конфігурації.

12 листопада 2024 року Microsoft випустила відкриту платформу .NET 9. Проект створений завдяки уніфікації продуктів .NET Framework, .NET Core та Mono. На основі рішення .NET 9 можна створювати багатоплатформні програми для браузера, хмарних систем, робочого столу, IoT-пристроїв та мобільних платформ, використовуючи єдині бібліотеки та загальний процес складання, який не залежить від типу програми. Пов'язані з проектом напрацювання, включаючи .NET Runtime та SDK, опубліковані на GitHub.

Збірки .NET SDK 9, .NET Runtime 9 та ASP.NET Core Runtime 9 сформовані для Linux, macOS та Windows. .NET Desktop Runtime 9 постачається лише для Windows. До складу .NET 9 входить Runtime з JIT-компілятором RyuJIT, специфікації API, бібліотеки WPF, Windows Forms, WinUI та Entity Framework, інтерфейс командного рядка dotnet, а також інструменти для розробки мікросервісів, бібліотек, серверних, графічних та консольних програм.

Окремо поставляється стек для розробки веб-застосунків ASP.NET Core 9.0, а також ORM-шар Entity Framework Core 9.0 (драйвери є, в тому числі, для SQLite і PostgreSQL), бібліотека WPF 9 (Windows Presentation Foundation), фреймворк Windows Forms 9 для розробки GUI, платформа Aspire 9 для створення програм Cloud Native, фреймворк MAUI 9 для розробки багатоплатформних інтерфейсів користувача, а також випуски мов C# 13 та F# 9. Підтримка .NET 9.0 та C# 13 включена у вільний редактор коду Visual Studio Code [13].

1.3 Типи створюваних додатків

ASP.NET Core – це універсальний веб-фреймворк, який можна використовувати для створення різних програм й веб-додатків. Він включає API, які підтримують безліч парадигм:

а) Razor Pages – використовується для створення багатосторінкових додатків з відображенням на стороні сервера [14];

б) архітектурний шаблон MVC – аналогічний Razor Pages й призначений для серверних програм, але без сторінкової парадигми [15, 16];

в) Blazor WebAssembly – фреймворк для створення односторінкових додатків на основі браузера, що використовує стандарт WebAssembly. Це аналог таких JavaScript фреймворків, як Angular, React та Vue [17];

г) Blazor Server – використовується для створення додатків з відстеженням стану та промальовуванням на стороні сервера, які відправляють події користувальницького інтерфейсу та оновлення сторінок через WebSockets, щоб забезпечити відчуття однієї сторінкової програми на стороні клієнта, але з легкістю розробки програми з відтворенням на стороні сервера.

д) Інтерфейси програмування застосунків (API):

– мінімальні API – прості API для протоколу HTTP, які можуть використовуватися мобільними програмами або односторінковими браузерними програмами;

– веб-API – це альтернативний підхід до створення API для протоколу HTTP, який додає більше структури та функціональних можливостей порівняно з мінімальними API [18];

– gRPC API – використовуються для створення ефективних двійкових API для обміну даними між серверами з використанням протоколу gRPC.

Розглянемо детальніше популярні типи на прикладі простого додатку, який генерує й відображає випадкове число грального кубика:

1.3.1 Шаблон MVC

Архітектурний шаблон (патерн) «модель-вигляд-контролер» (MVC) був придуманий в 70-х рр. минулого століття. Він з'явився в додатках із графічним інтерфейсом, але став дуже популярним при розробці веб-додатків. Написання HTML і CSS, відповідальних за зовнішній вигляд сторінки, – це зовсім інша навичка, ніж реалізація серверної частини. Тому відділення користувацького інтерфейсу від логіки має сенс, і MVC – один з доступних варіантів. Будучи адаптованим під веб-додатки, MVC працює наступним чином (рисунок 1.2):

– контролер приймає користувацький ввід (у випадку з веб-додатком – це дані HTTP-Запиту);

– контролер отримує модель (переважно, дані з бази даних) й маніпулює нею, а потім передає цю модель вигляду (зазвичай, HTML сторінці);

– клієнт отримує вигляд й може використовувати його для створення нового запиту.

В ASP.NET MVC ці компоненти, зазвичай, представлені наступним чином:

а) контролер – це клас C#. Запити зіставляються з методами дії. По суті, це загальнодоступні методи C#;

б) модель зазвичай являє собою об'єкт або клас C#, часто заповнений вмістом бази даних (але не обов'язково). У прикладах Microsoft переважно використовують Entity Framework Core, інструмент об'єктно-реляційного відображення, що, звичайно ж, не є обов'язковим. Контролер одержує доступ до моделі й може маніпулювати нею, а потім при необхідності передає її вигляду;

в) вигляд – це, по суті, HTML сторінка з додатковою розміткою для прив'язки значень із моделі або виконання коду. Оскільки ми використовуємо мову C#, ці сторінки мають розширення *cshtml*. Рушій виглядів Razor дозволяє включати код C# у ці файли за допомогою спеціального символу @. Файли компілюються, даючи можливість виконати C# код. Браузер, зрозуміло, одержує фінальний HTML код.

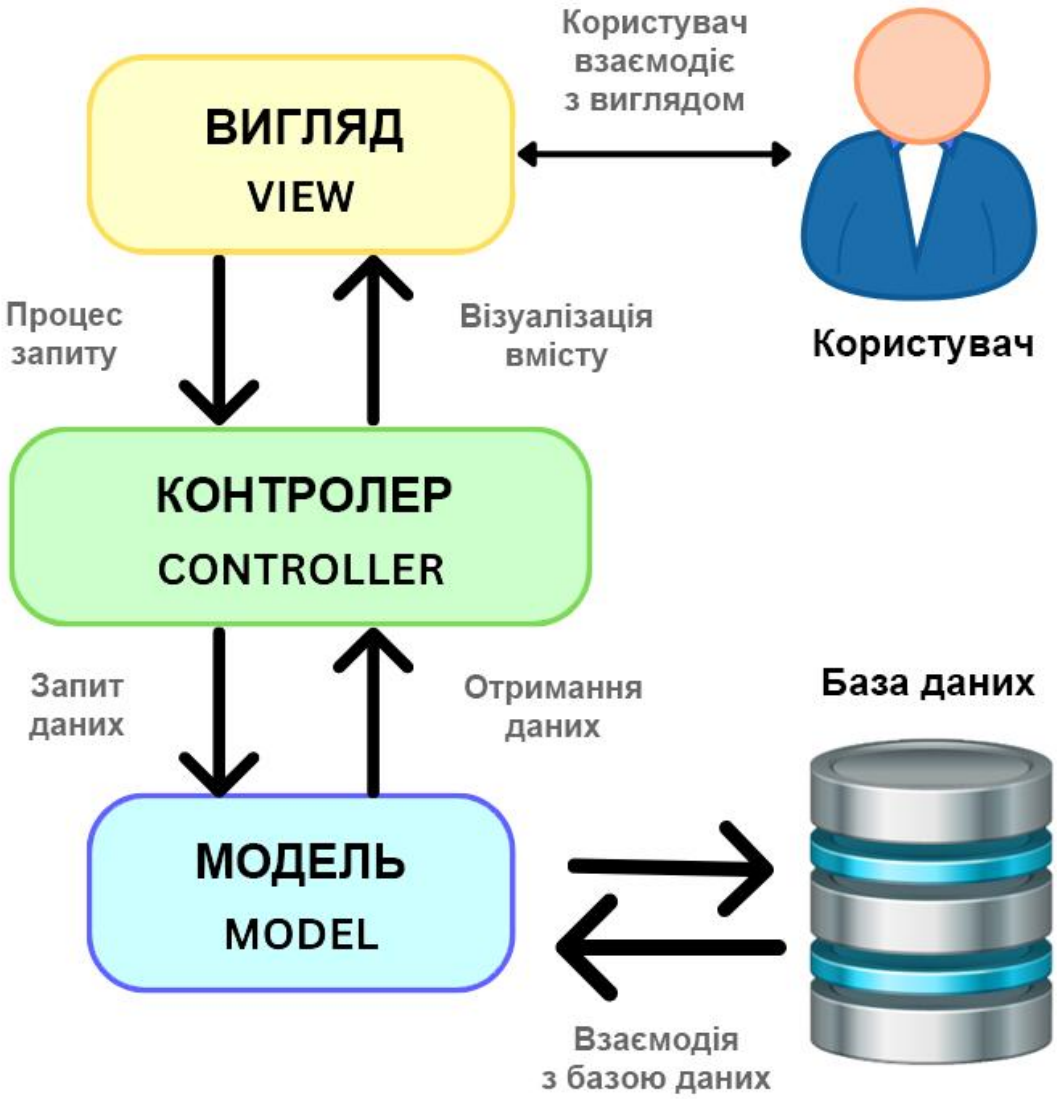


Рисунок 1.2 – Принцип роботи патерну MVC

Розглянемо основні елементи простого додатку шаблону MVC. На рисунку 1.3 показано лістинг контролеру даного додатку.

Клас HomeController реалізує метод дії Index(), який повертає вигляд з результатом випадкового числа кидка кубика. Тип DiceRoll визначений у тому ж файлі винятково для простоти.

Лістинг вигляду додатку MVC показано на рисунку 1.4. У вигляді в результаті кидка кубика властивість Outcome відображається в елементі <h1>.

HomeController.cs

WebApp — WebApp.Controllers.Hor

```
1   using Microsoft.AspNetCore.Mvc;
2
3   namespace WebApp.Controllers
4   {
        Ссылок: 3
5       public class HomeController : Controller
6       {
7           private readonly ILogger<HomeController> _logger;
8
            Ссылок: 0
9           public HomeController(ILogger<HomeController> logger)
10          {
11              _logger = logger;
12          }
13
            Ссылок: 0
14          public IActionResult Index()
15          {
16              var Outcome = new Random().Next(1, 7);
17              var Roll = new DiceRoll(Outcome);
18              return View(Roll);
19          }
20      }
21
            Ссылок: 1
22      public record DiceRoll(int Outcome);
23  }
24
```

Рисунок 1.3 – Лістинг контролера додатку MVC

Index.cshtml

WebApp

```
1   @{
2       ViewData["Title"] = "Кидок кубика - MVC";
3   }
4
5   <div class="text-center">
6       <h1>Випадкове число: @Model?.Outcome</h1>
7   </div>
8
```

Рисунок 1.4 – Лістинг вигляду додатку MVC

1.3.2 Razor Pages

Простий, але ефективний MVC синтаксис був покращений, щоб отримати власний підхід до веб-розробки в рамках ASP.NET Core. По суті, Razor Pages – це HTML сторінки з розширенням cshtml, які підтримують синтаксис Razor. На відміну від фреймворку MVC, тут немає необхідності в контролері. Весь код, відповідальний за отримання вигляду й обробку користувацького вводу тепер є частиною сторінки. Для додатків із простою логікою взаємодії користувацького інтерфейсу й коду для сторінки така архітектура є набагато зручнішою й зменшує складність, властиву архітектурі MVC. На рисунку 1.5 подано лістинг моделі додатку.

Index.cshtml.cs

WebApp — WebApp.Pages.IndexMo

```
1   using Microsoft.AspNetCore.Mvc.RazorPages;
2
3   namespace WebApp.Pages
4   {
        Ссылок: 8
5       public class IndexModel : PageModel
6       {
7           private readonly ILogger<IndexModel> _logger;
8
            Ссылок: 0
9           public IndexModel(ILogger<IndexModel> logger)
10          {
11              _logger = logger;
12          }
13
            Ссылок: 0
14          public void OnGet()
15          {
16              var Outcome = new Random().Next(1, 7);
17              Roll = new DiceRoll(Outcome);
18          }
19
            Ссылок: 2
20          public DiceRoll? Roll { get; set; }
            Ссылок: 2
21          public record DiceRoll(int Outcome);
22      }
23  }
24
```

Рисунок 1.5 – Лістинг моделі додатку Razor Pages

У цьому випадку, при відсутності окремого контролера, такий клас IndexModel уже містить код, що виконує кидок кубика при завантаженні сторінки. Модель, у свою чергу, прив'язана до вигляду. Вигляд Razor (рисунок 1.6) виглядає майже так само, як вигляд з додатка MVC (рисунок 1.4).

Index.cshtml

WebApp

```
1   @page
2   @model IndexModel
3   @{
4       ViewData["Title"] = "Кидок кубика - Razor Page";
5   }
6
7   <div class="text-center">
8       <h1>Випадкове число: @Model?.Roll?.Outcome</h1>
9   </div>
10
```

Рисунок 1.6 – Лістинг вигляду додатку Razor Pages

1.3.3 Веб-API

За допомогою ASP.NET Web API компанія Microsoft надає фреймворк для реалізації Rest сумісних API [18]. Створити свою реалізацію кінцевих точок API при цьому досить просто – достатньо взяти дані з бази даних, перетворити їх у формат JSON або XML й повернути клієнтові. При цьому значну частину цієї роботи веб-API виконує за розробника:

– залежно від значення заголовку HTTP запиту Accept використовується відповідний формат (наприклад, JSON або XML);

– керування версіями API здійснюється на основі HTTP заголовків, компонентів шляху або рядка запиту;

– коректно форматуються повідомлення про помилки й вийнятки;

– спрощується повернення правильного коду стану HTTP.

З погляду розробки, веб-API працює аналогічно ASP.NET MVC, відрізняються лише кілька базових класів. Зазвичай, веб-API працює швидше, оскільки Razor Pages не використовується. По суті, ми працюєте із класом,

який виглядає точно так само, як контролер, і з методами, які поводяться і виглядають, як методи MVC. На рисунку 1.7 показано лістинг контролера веб-API додатку. Контролер – це клас із методом Get(), який буде виконуватися при відправленні Get запиту до зв'язаної кінцевої точки. Дані, що повертаються, автоматично перетворяться у формат JSON.

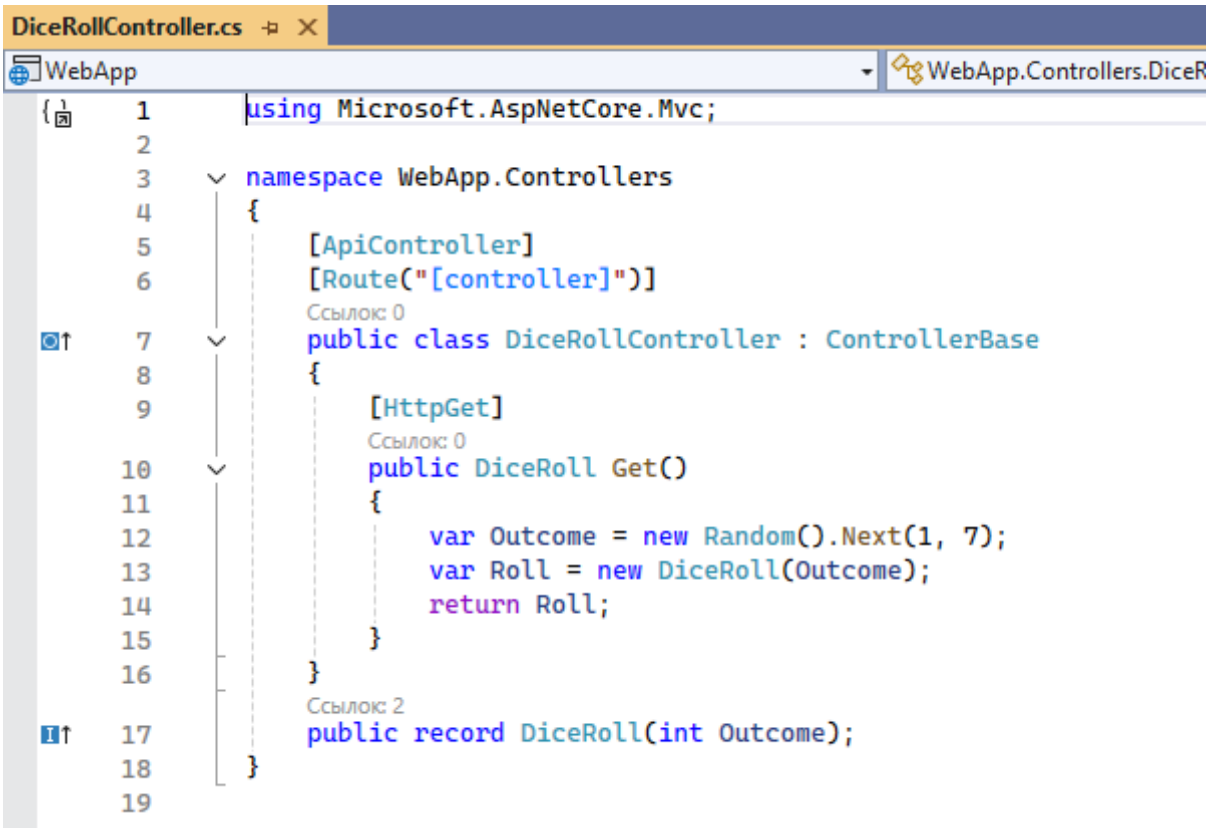


Рисунок 1.7 – Лістинг контролера веб-API додатку

1.3.4 Blazor

Із випуском Blazor Microsoft пробує інший підхід, цього разу орієнтований винятково на розробників, що не дуже люблять JavaScript. Усі сучасні браузери підтримують стандарт WebAssembly, що визначає двійковий формат коду, виконуваного в браузері. За допомогою Blazor код C# компілюється в WebAssembly, який потім може виконати браузер. Отже, можна писати додатки на C# (або інших мовах .NET), а браузер може їх запускати.

На рисунку 1.8 показана сторінка Blazor додатку. У даному випадку кидання кубика й вивід результату перебувають в одному файлі. У коді використовується синтаксис Razor із символом @.

Home.razor

BlazorApp

```
1   @page "/"
2
3   <PageTitle>Кидок кубика - Blazor</PageTitle>
4
5   @{
6       var Outcome = new Random().Next(1, 7);
7   }
8
9   <h1>Випадкове число: @Outcome</h1>
10
```

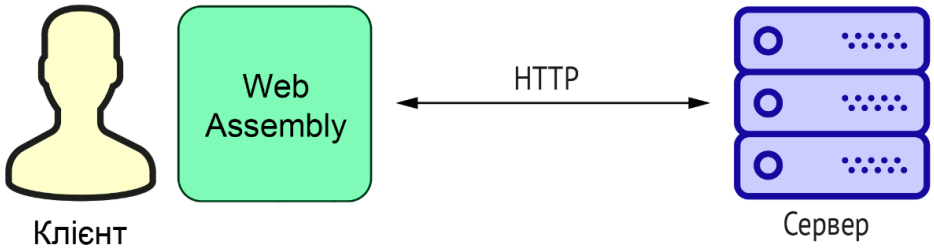
Рисунок 1.8 – Лістинг сторінки Blazor додатку

У даний час Blazor підтримує два підходи:

– Blazor Server – у браузер відправляється лише користувацький інтерфейс, тоді як весь код C# перебуває на веб-сервері. Згенерований код JavaScript автоматично викликає сервер (використовуючи SignalR), який потім виконує код C# і відправляє всі зміни в DOM (об'єктна модель документа – по суті, вміст) сторінки назад клієнтові. Додаток завантажується швидше, але не працює офлайн (рисунок 1.9, а);

– Blazor WebAssembly – усе компілюється в WebAssembly, включаю частини .NET, використовувані додатком. У цьому випадку завантаження додатка може зайняти кілька додаткових секунд, але потім він запускається в браузері без якої-небудь взаємодії із сервером, за винятком можливих викликів API. Як наслідок, код додатку можна піддати реверсу-інжинірингу (рисунок 1.9, б).

а)



б)

Рисунок 1.9 – Дві архітектурні моделі Blazor

а – Blazor Server; б – Blazor WebAssembly

1.4 Компоненти та загрози типового веб-додатку

При створенні нового проекту в Visual Studio обраний шаблон задає технологічний стандарт для веб-додатку. Вище зазначалося, що ASP.NET Core надає багато можливостей для створення сайтів: різні серверні фреймворки й підходи, а також різні методи й формати для клієнтського коду. По суті, велика кількість компонентів приводить до великої кількості місць, де щось може піти не так з погляду безпеки [19].

При цьому веб-додаток може взаємодіяти з іншими зовнішніми ресурсами, наприклад (рисунок 1.10):

– бази даних (Database);

– ресурси файлової системи;

– інтерфейси програмування застосунків (API);

– різноманітні зовнішні сервіси;

– мережі доправлення контенту (CDN), які містять бібліотеки чи інші, переважно статичні, ресурси.

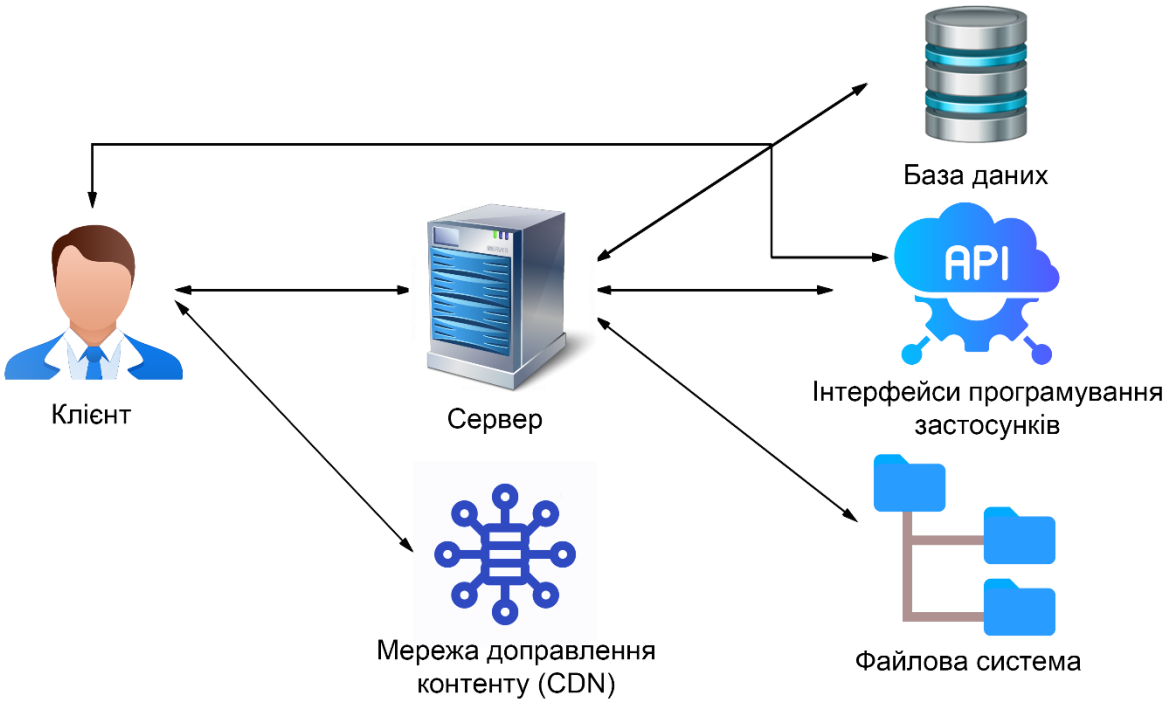


Рисунок 1.10 – Взаємодія веб-додатку із зовнішніми ресурсами

З погляду безпеки більша частина компонентів може виявитися під загрозою, у тому числі й зв'язки між ними [20]. Ось деякі загрози, від яких веб-додаток повинен забезпечувати захист:

– користувацькі дані, що відправляються на сервер, а згодом – клієнтові, можуть містити небажаний контент, наприклад шкідливий код JavaScript. Цей тип вразливості називають міжсайтовим скриптінгом (XSS);

– користувацький дані не проходять необхідної перевірки, при цьому вони відправляються в базу даних й можуть містити шкідливі команди, які потім виконуються в цій базі. Така атака називається SQL ін'єкцією;

– механізм проходження автентифікації в інтернеті може бути використаний для створення HTTP запитів, обробка яких може приводити до виконання певних дій від імені користувача, де він автентифікований. Типовою атакою цього типу є міжсайтова підробка запитів (CSRF);

– недостатня авторизація може надати користувачам доступ до ресурсів сервера, яких вони не повинні бачити;

– неочікуване ведення нетипових даних користувачем (наприклад,

занадто великих або невірних) приводять до небажаної поведінки веб-додатка або провокує появу повідомлень про помилки;

– дані, якими обмінюються клієнт і сервер, можуть бути перехоплені або викрадені, що піддає ризику деякі частини додатку. На допомогу приходить використання захищеного каналу обміну даними;

– ресурси, розміщені на сторонньому сайті (наприклад, CDN), можуть бути змінені;

– повідомлення про помилки можуть розкривати внутрішню інформацію про сервер, яка може бути корисна зловмисникові. Тому внутрішні помилки необхідно правильно обробляти й протоколювати.

1.5 Ешелонований захист веб-додатків

Якщо говорити про безпеку веб-додатків, то тут існує емпіричне правило: якщо щось може піти не так, то так і буде. От чому потрібно передбачити всі вищезгадані ризики й багато інших факторів, використовувати захисне програмування, при якому код додатка пишеться, чекаючи гіршого сценарію й реалізується якнайбільше заходів щодо забезпечення безпеки.

Іноді, коли справа стосується заходів безпеки, надмірність не буде зайвою. Краще використовувати два засоби захисту, ніж один (або не використовувати взагалі). Крім того, можна використовувати заходи безпеки на різних рівнях додатку. Якщо мова заходить про продуктивність, надлишковість заходів часто вважається дещо поганою. Навіщо робити дві речі, якщо досить однієї? З іншого боку, коли справа доходить до доступності, надлишковість допомагає підтримувати роботу додатку: якщо одна система захисту вийде з ладу, є інша. Безпека веб-додатків більш тісно пов'язана з доступністю, тому надлишкові заходи не є проблемою, зазвичай, вони вітаються.

Як правило, ми говоримо про механізми ешелонованого захисту при впровадженні декількох рівнів безпеки. Навіть якщо один з рівнів виявиться недостатнім, ми сподіваємося, що є й інші, які запобіжать збою додатку. Будь-який захід, спрямований на забезпечення безпеки, може дати збій, тому наявність ще одному такого заходу може бути надзвичайно корисною. Надалі в роботі ми зустрінемося з механізмами, які забезпечують додатковий рівень безпеки, але самі по собі вони не здатні повністю нейтралізувати загрози. Однак у комбінації з іншими засобами веб-додаток може стати щитом проти атак.

1.6 API, призначені для виконання функцій безпеки

Найбільш важливим аспектом безпеки веб-додатків є нейтралізація ризиків шляхом прогнозування атак та реалізації підходящих заходів протидії. ASP.NET Core та пов'язані з ним технології мають ряд API, які призначені для виконання функцій безпеки, що не завжди можна прямо зіставити з конкретною атакою. Зокрема:

– ASP.NET Core Identity надає API для перевірки автентичності й авторизації у веб-додатку, включаючи вхід/вихід користувача, дані профілю, ролі й багато чого іншого [21];

– HTTP заголовки, призначені для виконання функцій безпеки, можна налаштувати явно у файлі web.config та контролерах або неявно шляхом визначення параметрів у класі Program;

– безпечну взаємодію можна забезпечити різними варіантами й підходами, включаючи перенаправлення на HTTPS, захистом файлів cookie та навіть відключенням HTTP;

– паролі не повинні зберігатися у вигляді відкритого тексту. Вони повинні бути зашифровані, а ще краще, якщо вони будуть хешовані. В ASP.NET Core є відповідні формати й алгоритми для цього;

– у хмарних провайдерів, зокрема Microsoft Azure або Amazon Web

Services (AWS) є власні API для зберігання строк підключення або паролів.

Дуже важливо знати ці API а також філософію й механізми безпеки ASP.NET. Однак на практиці основною причиною є фактичні вразливості в коді, і аудит інформаційної безпеки підтверджує це знову й знову.

Висновки до розділу 1

1. Переважна більшість веб-додатків мають ті чи інші вразливості, значна частина яких обумовлена недосконалістю коду.

2. Більшість атак не залежать від конкретної використовуваної серверної технології. Переважно вони використовують спільні технології усіх веб-додатків: JavaScript, HTML, HTTP, CSS та ін.

3. Ризику атак також піддаються інші частин веб-додатку: клієнтська частина, сервер, база даних, зовнішні ресурси, шляхи взаємодії між ними.

4. Фреймворк ASP.NET Core має вбудовані засоби захисту: частина з них активована за замовчуванням, інша – готова до налаштування й активації, для решти – потрібно писати власний код.

2 МЕТОДИ ЗАХИСТУ ASP.NET CORE ВЕБ-ДОДАТКІВ

2.1 Захист від міжсайтового скриптингу

Міжсайтовий скриптинг (XSS) – це тип атаки, який дозволяє запускати код у браузері іншого користувача. Найбільш поширений різновид міжсайтового скриптингу – впровадження коду JavaScript на сторінку, хоча є й різновиди атак, які використовують HTML або CSS [22]. Впроваджений код JavaScript вважається частиною сторінки, що атакується, і виконується з тими ж привілеями, що і наш власний код, – він використовує той же контекст безпеки [23].

Ось кілька варіантів щодо того, на що здатний JavaScript:

– перенаправлення користувача на інший сайт (фішинг);

– зміна HTML, щоб форма входу надсилала дані на сторонній сайт (фішинг);

– додавання нових HTML елементів, наприклад, підроблена форма входу (фішинг);

– крадіжка файлів cookie;

– перехоплення натискань клавіш клавіатури (кейлоггінг);

– додавання великої кількості нових HTML елементів, що сильно навантажують браузер (відмова в обслуговуванні);

– видобуток криптовалют з використанням процесора жертви;

– експлуатація вразливостей у веб-браузерах або плагінах (при умові, що вони ще використовуються).

Цей список не є вичерпним, але він дає чітке уявлення щодо того, чому варто уникати міжсайтового скриптингу.

На рисунку 2.1 показано базовий приклад XSS атаки. Звичайні користувачі вашого додатку можуть надіслати своє ім'я до вашої програми, заповнивши форму. Потім програма додає ім'я у внутрішній список й відображає весь список на сторінці. Якщо імена не відображаються безпечно, зловмисник може виконати JavaScript в браузері будь-якого іншого

користувача, який переглядає список.

Користувач вводить фрагмент HTML замість свого імені. Коли інші користувачі переглядають список імен, шаблон Razor відображає імена за допомогою методу @Html.Raw(), який записує тег <script> у документ. Введені дані будь-якого користувача стають частиною HTML-розмітки сторінки для інших користувачів. Щойно сторінка завантажується в браузері нового користувача, тег <script> виконується й комп'ютер користувача виявляється скомпрометованим. Як тільки зловмиснику вдається виконати довільний код JavaScript у браузері звичайного користувача, він зможе робити практично все, що завгодно.

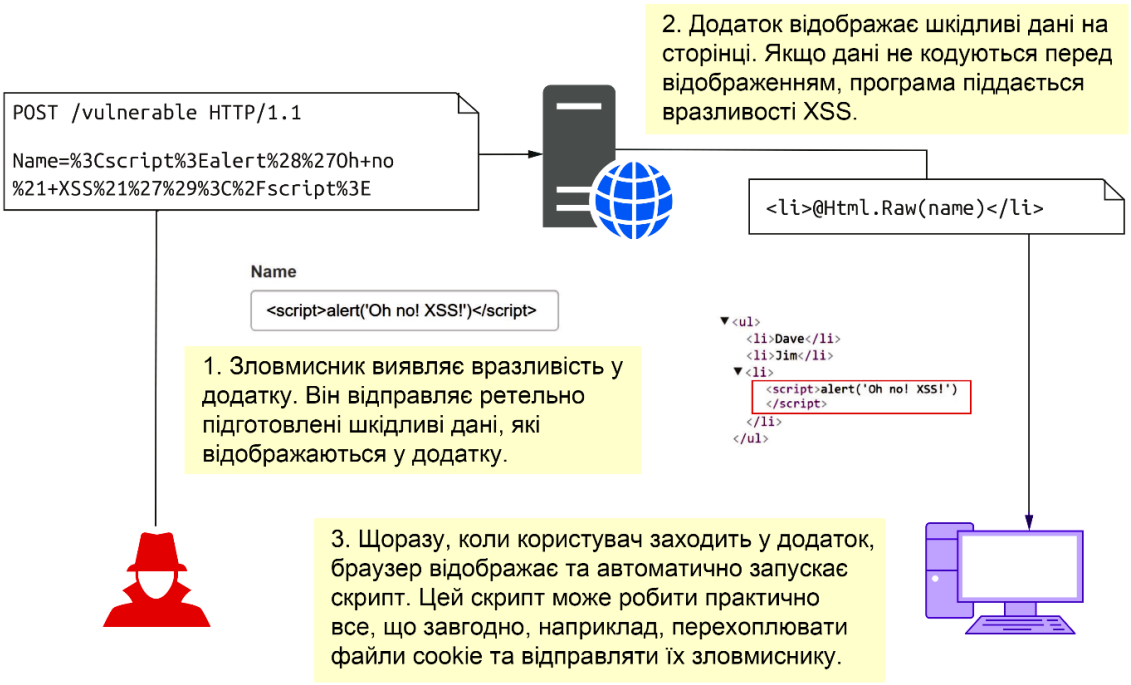


Рисунок 2.1 – Механізм експлуатації XSS вразливості

Вразливість тут пов'язана з небезпечним відображенням даних, які ввів інший користувач. Якщо дані не були закодовані, щоб зробити їх безпечними, перш ніж їх візуалізувати, комп'ютери інших користувачів будуть відкриті для атаки. За умовчанням Razor захищає від XSS-атак шляхом кодування всіх записаних даних в HTML за допомогою тегхелперів, HTML-хелперів або

синтаксису @ (рисунок 2.2). Тег <script> у цьому прикладі закодований, тому він більше не відображається як HTML і не може бути використаний для компрометації програми.

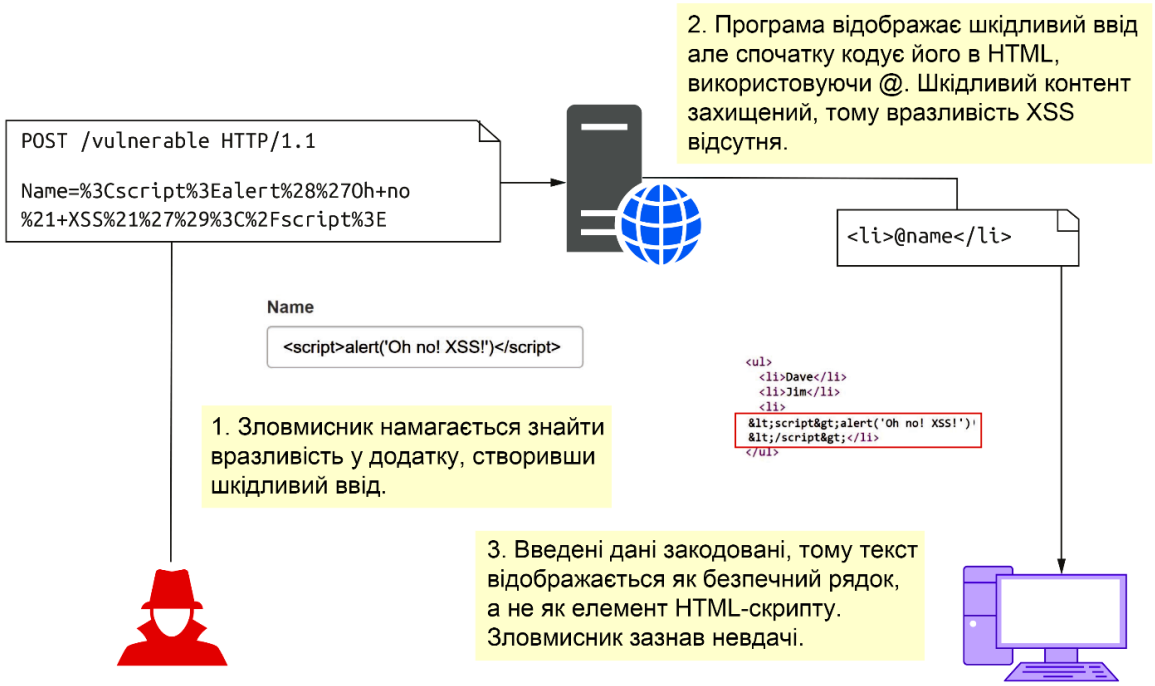


Рисунок 2.2 – Захист від XSS атаки за допомогою HTML-кодування введеної користувачем інформації з використанням символу @ у шаблонах Razor

Даний приклад демонструє використання кодування HTML для запобігання безпосереднього додавання елементів HTML DOM, але це не єдиний випадок, про який варто пам'ятати. Якщо ми передаємо неперевірені дані в JavaScript або використовуємо такі дані в значеннях URL-запитів, необхідно переконатися, що вони правильно кодовані.

Найпоширеніший приклад, це коли на сторінках Razor використовується jQuery або JavaScript і ми хочемо передати значення із сервера клієнту. Якщо використати стандартний символ @ для відображення даних на сторінці, вивід буде у кодуванні HTML. На жаль, якщо ми кодуємо рядок в HTML і вставляємо його безпосередньо в код JavaScript, то, ймовірно, не отримаємо того, чого очікуємо.

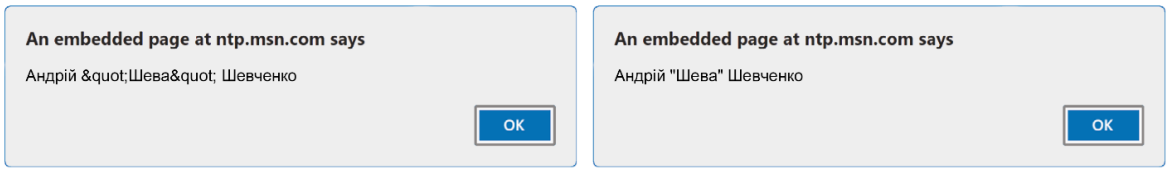
Наприклад, якщо у нашому файлі Razor є змінна з ім'ям name і ми хочемо зробити її доступною в JavaScript, у нас може виникнути думка реалізувати це так:

```
<script>var name = '@name'</script>
```

Якщо ім'я містить спеціальні символи, Razor закодує їх, використовуючи HTML кодування, а це не те, що потрібно в цьому контексті JavaScript. Наприклад, якби у змінної `name` було значення Андрій "Шева" Шевченко, тоді при відображенні, ми отримали б наступне:

```
<script>var name = 'Андрій &quot;Шева&quot; Шевченко';</script>
```

Зверніть увагу, що подвійні лапки (") були закодовані HTML як ". Якщо ми використовуємо це значення в коді JavaScript напряму, очікуючи, що воно буде «безпечним» закодованим значенням, воно буде виглядати неправильно (рисунок 2.3).



а) б)

Рисунок 2.3 – Порівняння попереджень під час використання кодування JavaScript та кодування HTML

а – при використанні HTML-кодування лапки у JavaScript відображаються некоректно; б – кодування JavaScript забезпечує безпечний спосіб відображення користувацького введення в очікуваному форматі

Замість цього потрібно закодувати змінну, використовуючи кодування JavaScript, щоб подвійні лапки відображалися як безпечний символ Юнікоду, \u0022. Цього можна досягти за допомогою впровадження JavaScriptEncoder у вигляд та виклику методу Encode() для змінної name:

```
@inject System.Text.Encodings.Web.JavaScriptEncoder encoder;
    <script>var name = '@encoder.Encode(name)'</script>
```

Щоб уникнути необхідності пам'ятати про використання кодування JavaScript ми рекомендуємо не записувати значення JavaScript таким чином. Натомість краще записати значення в атрибути HTML елементу, а згодом передайте його в змінну JavaScript дещо пізніше. Це повністю звільняє від необхідності використовувати кодувальник JavaScript (рисунок 2.4).

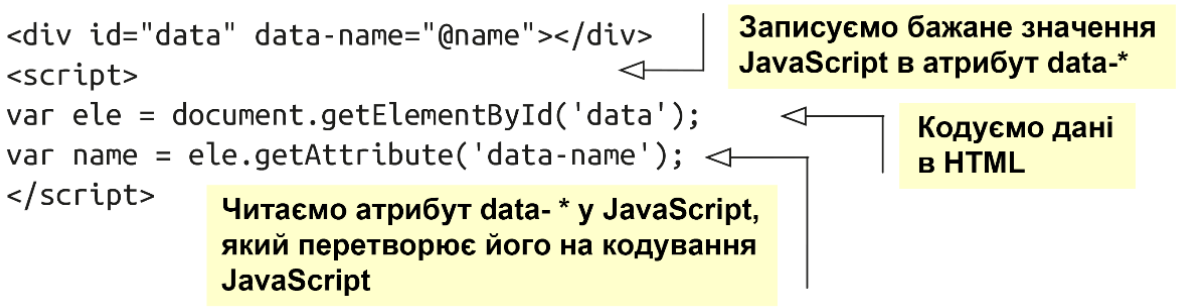


Рисунок 2.4 – Лістинг коду передачі значень JavaScript шляхом запису їх в атрибути HTML

Для запобігання XSS атак валідація вхідних даних іноді може допомогти, але часто не все так просто. Наприклад, простий валідатор імені може вимагати, щоб користувач використовував лише літери. Це дозволить запобігти більшості атак. На жаль, тут не враховуються користувачі, у яких в імені є дефіс або апостроф, не кажучи вже про користувачів з незахідними іменами. Користувачі зі зрозумілих причин засмучуються, коли додаток їм повідомляє, що їхнє ім'я недійсне. Варто остерігатися такого підходу. Незалежно від того, чи використовується сувора валідацію чи ні, дані завжди повинні кодуватися, коли вони візуалізуються на сторінці. Варто добре подумати, коли використовується @Html.Raw(). Якщо у користувача є спосіб введення шкідливих даних у це поле, варто знайти інший спосіб відображення даних.

2.2 Захист від міжсайтової підробки запиту

Міжсайтова підробка запитів (CSRF) може бути проблемою для веб-сайтів або API, які використовують файли cookie для автентифікації. Така атака здійснюється з шкідливого веб-сайту, що створює автентифікований запит до вашого API від імені користувача, причому користувач його не ініціював.

Класичний приклад такої атаки – банківський переказ чи виведення коштів. Уявіть, що ми маємо банківський додаток, який зберігає токени автентифікації у файлах cookie, як це зазвичай буває (особливо у традиційних серверних додатках). Браузери автоматично надсилають файли cookie, асоційовані з доменом, з кожним запитом, щоб програма знала, чи користувач пройшов автентифікацію.

Тепер уявимо, що у вашому додатку є сторінка, яка дозволяє користувачеві переказувати кошти зі свого рахунку на інший рахунок за допомогою запиту POST на сторінку Balance. Вам потрібно виконати вхід, щоб отримати доступ до форми (ми захистили сторінку Razor за допомогою атрибуту [Authorize]), а після цього ми просто надсилаємо форму, використовуючи POST запит, в якому вказана сума, яку ми хочемо переказати, із зазначенням того, куди ми хочемо переказати гроші.

Припустимо, користувач заходить на наш сайт, виконує вхід і транзакцію. Потім він відвідує інший сайт, який контролює зловмисник. Зловмисник додав форму введення у свій сайт, який виконує POST запит на сайт вашого банку, ідентичний тому, що використовувався для форми на сайті банку, щоб переказати кошти. Ця форма робить деякі шкідливі дії, наприклад, переводить всі гроші користувача зловмиснику, як показано на рисунку 2.5. Браузери автоматично надсилають файли cookie для програми, коли сторінка виконує POST запит у повній формі, і банківська програма не може знати, що це зловмисний запит. Користувач, який нічого не підозрює, віддає всі свої гроші зловмиснику.

Вразливість тут викликана тим, що браузери автоматично відправляють файли cookie при запиті сторінки (з використанням GET запиту) або надсилання вмісту з форми. Немає різниці між легальним запитом POST для надсилання форми у вашому банківському додатку і таким же запитом, що виходить з боку зловмисника. На жаль, подібна поведінка вбудована у мережу – це те, що дозволяє без проблем подорожувати сайтами, після того як ми виконали вхід.

Порядок дій змовмисника при експлуатації CSRF вразливості зображено на рисунку 2.5. На шкідливому сайті вже є форма, яка відповідає формі у нашому додатку, і зловмисник відправляє її вміст у нашу програму. Браузер відправляє cookie-файл автентифікації автоматично, тому наша програма сприймає запит як дійсний запит від користувача

Рисунок 2.5 – Механізм експлуатації CSRF вразливості

Поширеним рішенням при такій атаці є синхронізуючий токен, який використовує унікальні токени верифікації, щоб забезпечити відмінність між легітимним запитом POST і підробленим запитом від зловмисника. Один токен зберігається у файлі cookie, а інший – додається у форму, яку потрібно

захистити. Наша програма генерує токени під час виконання на основі поточного користувача, який виконав вхід, тому зловмисник не зможе створити такий токен для своєї підробленої форми.

Браузери мають додаткові засоби захисту для запобігання відправленню файлів cookie у даній ситуації, які називаються файлами cookie SameSite. За замовчуванням більшість браузерів використовують SameSite=Lax, що запобігає цій атаці [24].

На рисунку 2.6 подано механізм захисту від CSRF-атак за допомогою токена верифікації. Коли зі сторінки Balance на сервер передано POST-запит, на сервері порівнюються значення, доступні на сервері, і значення файлу cookie. Якщо відсутнє значення або токени не збігаються, запит відхиляється. Якщо зловмисник створює POST запит, браузер відправляє токен з cookie токена, але в самій формі токена не буде або він буде недійсним. Сторінка Razor відхиляє запит, таким чином захищаючись від атаки (рисунок 2.6).

Рисунок 2.6 – Захист від CSRF-атак за допомогою токена верифікації

Хорошою новиною є те, що Razor Pages автоматично захищає нас від подібних атак. Тег хелпер форми автоматично встановлює файл cookie токена

верифікації та відображає токен у приховане поле RequestVerificationToken для кожного елемента <form> у нашому додатку (якщо ми спеціально не відключили їх). Наприклад, візьмемо цей простий шаблон Razor, який відправляється назад на ту саму сторінку Razor:

```
<form method="post">
<label>Amount</label>
<input type="number" name="amount" />
<butt on type="submit">Withdraw funds</button>
</form>
```

При відтворенні в HTML токен верифікації зберігається в прихованому полі і надсилається назад із справжнім запитом:

```
<form method="post">
<label>Amount</label>
<input type="number" name="amount" />
<button type="submit" >Withdraw funds</button>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Daz26qb0hBGsw7QCK"/>
</form>
```

ASP.NET Core автоматично додає ці токени у кожну форму, а Razor Pages автоматично перевіряє їх. Фреймворк гарантує, що токени є і у файлі cookie, і в даних форми, і гарантує їх з впадання і відхиляє будь-які запити у випадку, якщо вони не збігаються.

Якщо ми використовуємо контролери MVC з виглядами замість Razor Pages, то ASP.NET Core, як і раніше, додає токени верифікації у кожну форму. На жаль, він їх не перевіряє автоматично. Щоб увімкнути перевірку, потрібно додавати до контролерів спеціальний атрибут [ValidateAntiForgeryToken]. Це гарантує, що при роботі таких контролерів та виглядів для них у браузері відповідність токенів з cookie та на сервері перевірятиметься автоматично, і

якщо вони не збігатимуться, тоді сервер відхиляє такі запити [25].

Як правило, токени потрібно використовувати тільки для запитів із методами POST, DELETE та інших небезпечних типів запитів, які використовуються для зміни стану.

GET запити для цієї мети не використовуються, тому фреймворк не вимагає захисних токенів для їхнього виклику. Razor Pages перевіряє ці токени, якщо мова про небезпечні методи, такі як POST, і ігнорує безпечні методи, такі як GET. Поки ми створюємо свою програму, наслідуючи цей шаблон, фреймворк зробить все можливе, щоб ми були в безпеці. Якщо нам з якоїсь причини необхідно явно ігнорувати ці токени на сторінці Razor, ми можемо відключити перевірку, застосувавши атрибут [IgnoreAntiforgeryToken] у PageModel сторінок Razor. Це дозволяє обійти захист фреймворку для тих випадків, коли ми впевнені, що робимо щось безпечне, від чого не потрібно захист, але в більшості випадків краще перестрахуватися і перевірити все.

Все стає складніше, якщо ми робимо багато запитів до API за допомогою JavaScript і надсилаємо об'єкти у форматі JSON, а не дані форми. У таких випадках ми не зможемо відправити токен верифікації як частину форми (бо ми відправляєте JSON), тому нам потрібно буде замість цього додати його як заголовок у запит [25].

2.3 Створення унікальних токенів з API для захисту даних

Захисні токени, які використовуються для запобігання міжсайтовій підробці запитів, залежать від можливостей фреймворку використовувати сильне симетричне шифрування для шифрування та розшифрування даних. Алгоритми шифрування зазвичай покладаються на один або кілька ключів, які використовуються, щоб ініціалізувати шифрування та зробити цей процес відтворюваним. Якщо у нас є ключ, ми можемо шифрувати та розшифровувати дані; без цього ключа такі дані неможливо вкрасти.

В ASP.NET Core шифрування виконують API захисту даних (data protection API). Вони створюють захисні токени, шифрують cookie-файли автентифікації та генерують безпечні токени. Що особливо важливо – вони також контролюють керування файлами ключів, які використовуються для шифрування. Файл ключа – це невеликий XML-файл, який містить випадкове значення ключа, яке використовується для шифрування у програмах ASP.NET Core. Дуже важливо, щоб він зберігався в надійному місці: якщо зловмисник заволодіє ним, то зможе видавати себе за будь-якого користувача нашого додатку.

Система захисту даних зберігає ключі у надійному місці, залежно від того, як і де ми розмістили додаток:

а) якщо веб-додаток розміщено у хмарі Azure – у спеціальній синхронізованій папці, доступній усім регіонам;

б) якщо на сервері IIS без профілю користувача – зашифрованими реєстром;

в) якщо обліковий запис з профілем користувача – %LOCALAPPDATA%\ASP.NET\DataProtection-Keys у Windows або ~/.aspnet/DataProtection-Keys у Linux або macOS;

г) у всіх інших випадках – у пам'яті (при перезапуску додатку ключі будуть втрачені).

Щоб наш додаток міг читати cookie-файли автентифікації наших користувачів, він повинен мати можливість розшифрувати їх, використовуючи той самий ключ, який застосовувався для їхнього шифрування. Якщо ми працюємо з кількома серверами, то за замовчуванням кожен сервер має свій ключ, і ми не зможемо читати файли cookie, зашифровані іншими серверами. Щоб обійти цю проблему, потрібно налаштувати свою програму для зберігання ключів захисту даних в єдиному місці. Це може бути спільно використовувана папка на жорсткому диску, екземпляр Redis або, наприклад, екземпляр сховища BLOB-об'єктів Azure.

2.4 Безпека використання ресурсів з різних джерел

Як ми вже бачили, атаки з міжсайтовою підробкою запитів можуть бути досить потужними, але вони були б ще небезпечнішими, якби не браузери, що реалізують правило обмеження домену. Дана концепція забороняє програмам використовувати JavaScript для виклику вебAPI в іншому місці, якщо це не дозволено явно вебAPI.

Джерела вважаються однаковими, якщо вони відповідають протоколу (HTTP або HTTPS), домену (example.com) та порту (80 за промовчанням для HTTP і 443 для HTTPS). Якщо програма намагається отримати доступ до ресурсу за допомогою JavaScript та джерела не ідентичні, браузер блокує запит.

Правило обмеження домену є строгим: джерела двох URL-адрес повинні бути ідентичними, щоб запит був дозволений. Наприклад, наступні джерела однакові:

– http://example.com/home;

– http://example.com/site.css.

Шляхи цих двох адрес різняться (/home і /site.css), але протокол, домен і порт (80) ідентичні. Отже, якби ми перебували на головній сторінці свого додатка, то могли б без проблем запросити файл /site.css за допомогою JavaScript.

Навпаки, джерела наступних сайтів розрізняються, тому ми не можете запросити жодну з цих URL-адрес за допомогою JavaScript з джерела http://example.com:

– https://example.com – інший протокол (https);

– http: //www.example.com – інший домен (включаючи піддомен);

– http://example.com:5000 – інший порт.

Що стосується простих додатків, наприклад коли вся функціональність зосереджена в одному веб-додатку, таке обмеження може і не бути проблемою, але дуже часто додаток відправляє запити в інший домен.

Наприклад, у вас може бути сайт для онлайн-торгівлі, розташований за адресою http://shopping.com, і ми намагаємося завантажити дані з сайту http://api.shopping.com, щоб відобразити докладні відомості про товари, що є у продажу. При такій конфігурації ви зіткнетеся з правилом обмеження домену. Будь-яка спроба виконати запит за допомогою JavaScript до домену api.shopping.com завершиться помилкою, аналогічною до тієї, що показана на рисунку 2.7. Chrome заблокував запит від http://shopping.com:6333 до API за адресою http://api.shopping.com:5111. За замовчуванням браузер не дозволяє перехресні запити і блокує доступ нашого додатка до відповіді.

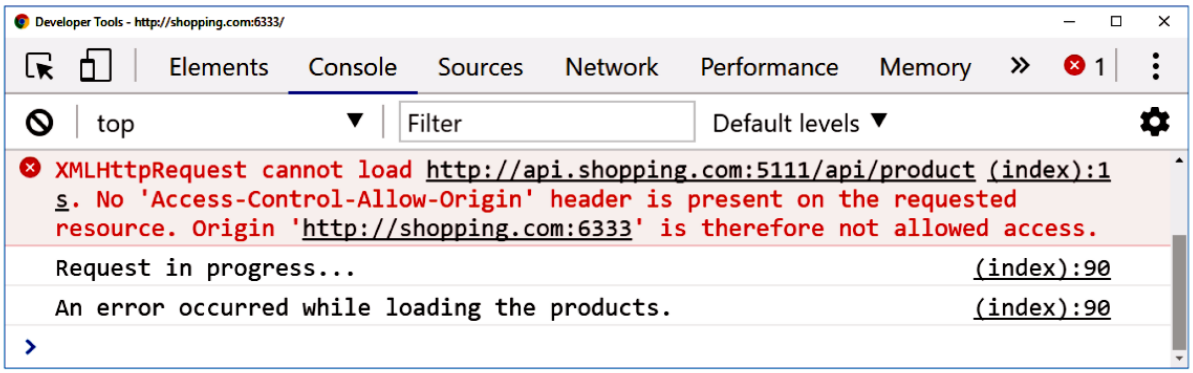


Рисунок 2.7 – Журнал консолі браузера Chrome для невдалого запиту

Необхідність виконувати запити, використовуючи різні джерела, JavaScript стає все більш поширеним явищем зі зростанням числа клієнтських односторінкових додатків і відходом від монолітних додатків. На щастя, є веб-стандарт, що дає змогу безпечно обійти цю проблему – це спільне використання ресурсів між різними джерелами (CORS).

CORS – це веб-стандарт, що дозволяє нашому вебAPI робити заяви про те, хто може виконувати запити з різних джерел. Ми можемо використовувати його, щоб контролювати, які програми можуть викликати ваш API, щоб мати можливість вирішувати сценарії, подібні до того, що описані вище.

Наприклад, можна:

– дозволити запити від http://shopping.com та https://app.shopping.com;

– дозволити запити з різних джерел лише якщо використовується метод GET;

– дозволити повертати заголовок Server у відповідях на запити з різних джерел;

– дозволити надсилання облікових даних (таких як cookie файли автентифікації або заголовки авторизації) із запитами з різних джерел.

Можна об'єднати ці правила в політику та застосовувати різні політики до різних кінцевих точок свого API. Ми можемо застосувати політику до всього додатку або окрему політику до кожної дії API.

CORS використовує HTTP заголовки. Коли наш веб-API отримує запит, він задає спеціальні заголовки для відповіді, щоб вказати, чи дозволені запити з різних джерел, що це за джерела і які HTTP методи та заголовки може використовувати запит – майже все, що стосується запиту.

У деяких випадках перед надсиланням реального запиту до нашого API браузер надсилає попередній запит. Він надсилається за допомогою методу OPTIONS, який браузер використовує, щоб перевірити, чи дозволено зробити цей запит. Якщо API поверне правильні заголовки, браузер надішле цей запит, як показано на рисунку 2.8.

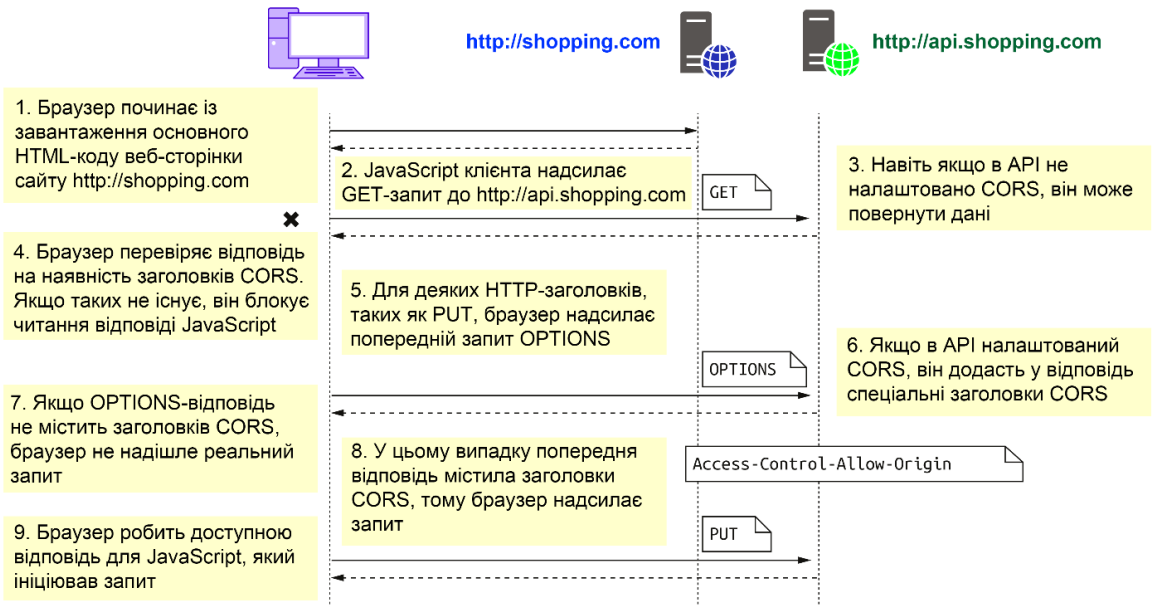


Рисунок 2.8 – Два запити з різних джерел

Відповідь на перший запит (рисунок 2.8) не містить заголовки CORS, тому браузер блокує їх читання додатком. Другий запит вимагає попереднього запиту за допомогою методу OPTIONS, щоб перевірити, чи активовано CORS. Оскільки відповідь містить заголовки CORS, можна виконати цей запит й надати відповідь JavaScript [26].

Браузери реалізують політику CORS з міркувань безпеки, тому слід уважно подумати про наслідки послаблення будь-яких обмежень, що накладаються ними. Навіть якщо ми активуємо можливість надсилання запитів з різних джерел, то все одно зможемо контролювати, які ці запити можуть надіслати і що поверне ваш API. У табл. 2.1 показано деякі доступні методи та результат їх дії.

Таблиця 2.1 – Методи, що реалізують політику CORS

Метод	Результат
WithOrigins("http://shopping.com")	Дозволяє запити з різних джерел від http://shopping.com.
AllowAnyOrigin()	Дозволяє запити з будь-якого джерела. Це означає, що будь-який веб-сайт може робити JavaScript-запити до API.
WithMethods() / AllowAnyMethod()	Задає дозволені HTTP-методи (такі як GET, POST та DELETE), за допомогою яких можна виконувати запити до вашого API.
WithHeaders()/ AllowAnyHeader()	Задає заголовки, які браузер може надсилати вашому API. Якщо ми обмежуємо заголовки, то повинні включити щонайменше заголовки "Accept", "Content-Type" та "Origin", щоб дозволити дійсні запити.
WithExposedHeaders()	Дозволяє нашому API надсилати до браузера додаткові заголовки. За замовчуванням у відповіді надсилаються лише заголовки Cache-Control, Content-Language, Content-Type, Expires, LastModified та Pragma.
AllowCredentials()	За замовчанням браузер не надсилає деталі автентифікації із запитами з різних джерел, лише якщо ми явно не дозволимо це. Ми також повинні активувати надсилання облікових даних на стороні клієнта JavaScript під час створення запиту.

Запити з різних джерел - лише один з безлічі потенційних способів, за допомогою яких зловмисники можуть скомпрометувати ваш додаток. Від багатьох з них легко захиститися, але потрібно знати про них і знати, як пом'якшити наслідки таких атак.

2.5 Виявлення та запобігання атакам з відкритим пере-направленням

Одна з найпоширеніших вразливостей, описана на сайті OWASP, пов'язана з атаками з відкритим перенаправленням. Це атака, при якій користувач клацає за посиланням, що веде на інший безпечний додаток, і зрештою потрапляє на шкідливий веб-сайт, наприклад, що обслуговує шкідливе ПЗ. При цьому, безпечний додаток не містить прямих посилань на шкідливий сайт. Розглянемо, як це відбувається.

Атаки з відкритим перенаправленням відбуваються, коли наступна сторінка передається в якості параметра методу дії. Найпоширеніший приклад - коли ми входимо в додаток. Зазвичай програми запам'ятовують сторінку, на якій знаходиться користувач, перш ніж перенаправити його на сторінку входу, передаючи поточну сторінку як параметр рядка запиту returnUrl. Після того, як користувач виконує вхід, програма перенаправляє користувача на returnUrl, щоб продовжити з того місця, де він зупинився.

Уявимо, що користувач переглядає сайт для онлайн-торгівлі. Він натискає кнопку «Купити» поруч із продуктом і перенаправляється на сторінку входу. Сторінка продукту, на якій він знаходився, передається у вигляді returnUrl, тому після входу він перенаправляється на сторінку продукту замість перекидання на головну сторінку.

Атака з відкритим перенаправленням використовує цей загально-прийнятий шаблон, як показано на рисунку 2.9. Зловмисник створює URL-адресу для входу, де для returnUrl задається значення у вигляді веб-сайту, на який він хоче відправити користувача. Зловмисник переконує користувача клацнути на посилання, що веде на ваш веб-додаток. Після того, як користувач

виконає вхід, вразлива програма перенаправить користувача на шкідливий сайт.

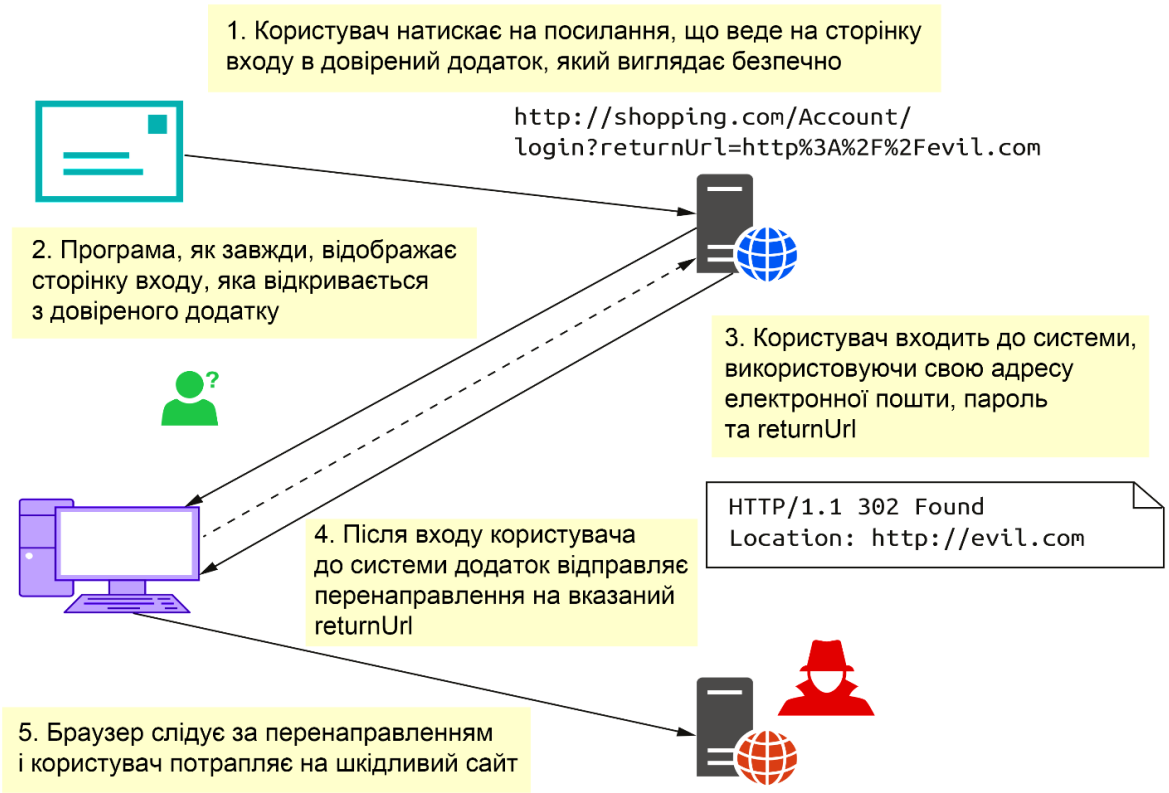


Рисунок 2.9 – Атака з відкритим перенаправленням

Простий спосіб вирішити таку проблему – завжди перевіряти, що returnUrl є локальною URL-адресою, що належить нашому додатку, до того, як перенаправити користувачів на нього. За замовчуванням інтерфейс Identity вже робить це, тому нам не потрібно турбуватися про сторінку входу.

Якщо у нас є перенаправлення в інших частинах програми, ASP NET Core надає кілька допоміжних методів для забезпечення безпеки, найбільш корисним є Url.IsLocalUrl(). У наступному лістингу (рисунок 2.10) показано, як переконатися, що наданий returnUrl безпечний, а якщо це не так, то виконати перенаправлення на головну сторінку програми. LocalRedirect() у класах ControllerBase і PageModel, який ініціює виняток, якщо надана URL-адреса не є локальною.

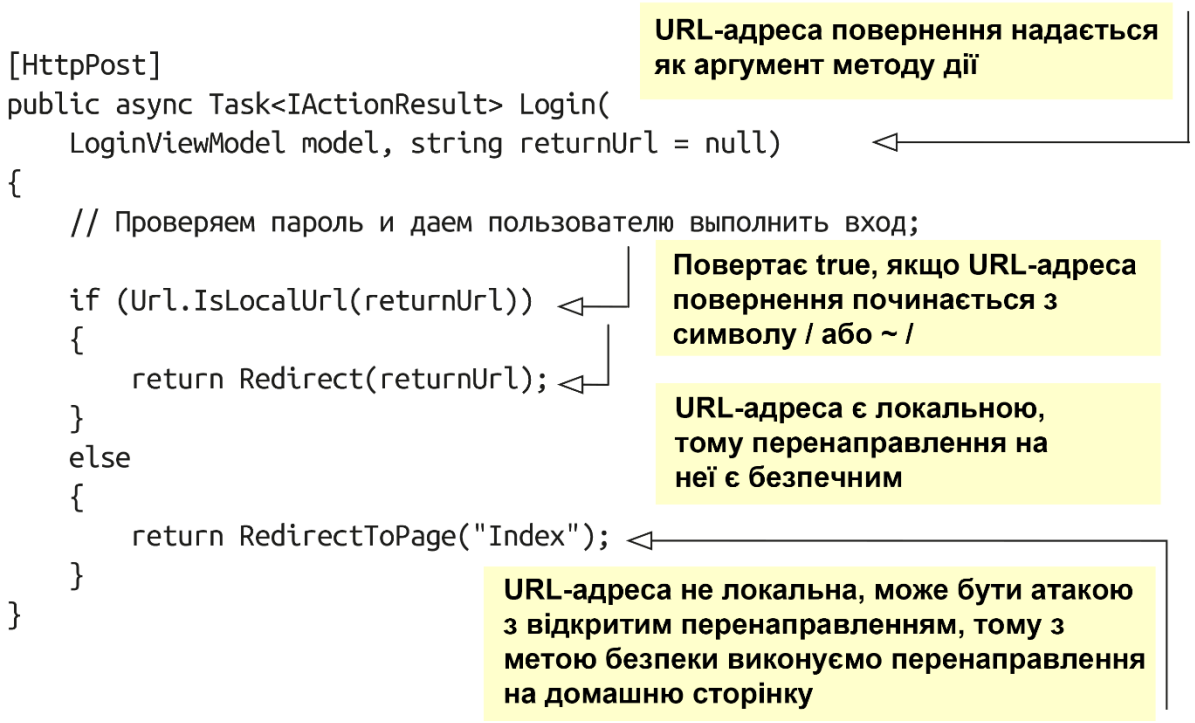


Рисунок 2.10 – Запобігання атакам з відкритим перенаправленням шляхом перевірки локальних URL адресів повернення

Цей простий шаблон забезпечує захист від атак відкритого перенаправлення, які в іншому випадку могли б надати нашим користувачам доступ до шкідливого контенту. Щоразу, коли ми перенаправляємо користувача на адресу, яка йде з рядка запиту або іншого введення користувача, ми повинні використовувати цей шаблон.

У деяких потоках автентифікації, наприклад при автентифікації за допомогою OpenID Connect, ми не можемо перенаправити на локальну URL-адресу, тому ми не можемо використовувати цей шаблон. Натомість OpenID Connect вимагає попередньої реєстрації дозволених URL-адрес перенаправлення та перенаправлення тільки на зареєстровану URL-адресу. Нам слід розглянути можливість використання цього шаблону, якщо ми не можемо забезпечити локальне перенаправлення.

Висновки до розділу 2

1. Міжсайтового скриптингу можна уникнути, якщо захищати небезпечний користувацький ввід перед його відображенням на сторінці. У Razor Pages це відбувається автоматично, якщо ми не застосовуємо метод @Html.Raw(). Тому даний метод варто використовувати його рідко та обережно.

2. Міжсайтова підробка запитів (CSRF) уникається шляхом використання автентифікації на основі файлів cookie, наприклад ASP.NET Core Identity. Дані атаки покладаються на те, що браузери автоматично відправляють файли cookie на веб-сайт. Шкідливий сайт може створити форму, яка надсилає запити за допомогою методу POST на ваш сайт, а браузер буде надсилати cookie файли автентифікації із запитом. Це дозволяє шкідливим веб-сайтам надсилати запити, начебто це був користувач, який виконав вхід.

3. Зменшити ризики CSRF атак можна, використовуючи захисні токени. Фреймворк Razor Pages автоматично додає захисні токени у будь-які форми, які ми створюємо за допомогою Razor, та перевіряє токени для вхідних запитів. При необхідності, перевірку валідації можна вимкнути.

4. Браузери не дозволяють веб-сайтам виконувати запити JavaScript між додатками з різних джерел. Щоб відповідати джерелу, програма повинна мати такий же протокол, домен та порт. Щоб зробити подібні запити з різних джерел, необхідно активувати спільне використання ресурсів між джерелами (CORS) у своєму додатку.

5. Захист від атак з відкритим перенаправленням можна реалізувати переконавшись, що виконується перенаправлення лише на локальні URL-адреси, що належать нашому додатку.

3 УДОСКОНАЛЕННЯ ЗАСОБІВ ЗАХИСТУ ВЕБ-ДОДАТКІВ

3.1 Налаштування політики CORS

3.1.1 Додавання глобальної політики CORS до всього додатку

Як правило, не слід налаштовувати CORS для своїх API, доки він вам не знадобиться. Браузери не просто блокують обмін даними між джерелами – так вони закривають шлях для багатьох атак. І поки у нас не з'явиться API в домені, відмінному від програми, якій необхідно отримати доступ до нього, не варто турбуватися про це. Щоб додати підтримку CORS в додаток, потрібні чотири речі:

– додати сервіси CORS у свою програму;

– налаштувати хоча б одну політику CORS;

– додати проміжне ПЗ CORS в конвеєр;

– задати політику CORS за промовчанням для всієї програми або декорувати дії веб-API атрибутом [EnableCors], щоб ви активувати CORS для певних кінцевих точок.

Щоб додати служби CORS до програми, необхідно викликати метод AddCors() для екземпляра WebApplicationBuilder з файлу Program.cs:

```
builder.Services.AddCors();
```

Більша частина зусиль з налаштування CORS йде на конфігурацію політики. Політика CORS контролює, як програма відповідатиме на запити з різних джерел. Вона визначає, які джерела дозволені, які заголовки повертати, які HTTP методи дозволити і т. д. Зазвичай політики визначаються одночасно з додаванням сервісів CORS у свою програму. Наприклад, розглянемо попередній приклад із сайтом для онлайн-торгівлі.

Для прикладу, ми хочемо, щоб наш API, розміщений на http://api.shopping.com, був доступний з основної програми через клієнтський JavaScript, розміщений на http://shopping.com. Тому потрібно налаштувати

API, щоб дозволити запити з різних джерел.

Варто пам'ятати, що саме основний веб-додаток буде отримувати помилки при спробі робити запити з різних джерел, проте CORS додається до API, до якого ми отримуємо доступ, а не до веб-додатку, який виконує запити.

На рисунку 3.1 показано лістинг, як налаштувати політику "AllowShoppingApp", щоб дозволити запити з різних джерел від http://shopping. com до API. Крім того, ми явно дозволяємо будь-який тип HTTP методу; без цього виклику дозволені лише прості методи (GET, HEAD та POST). Налаштування виконується з використанням шаблону проєктування Fluent Builder.

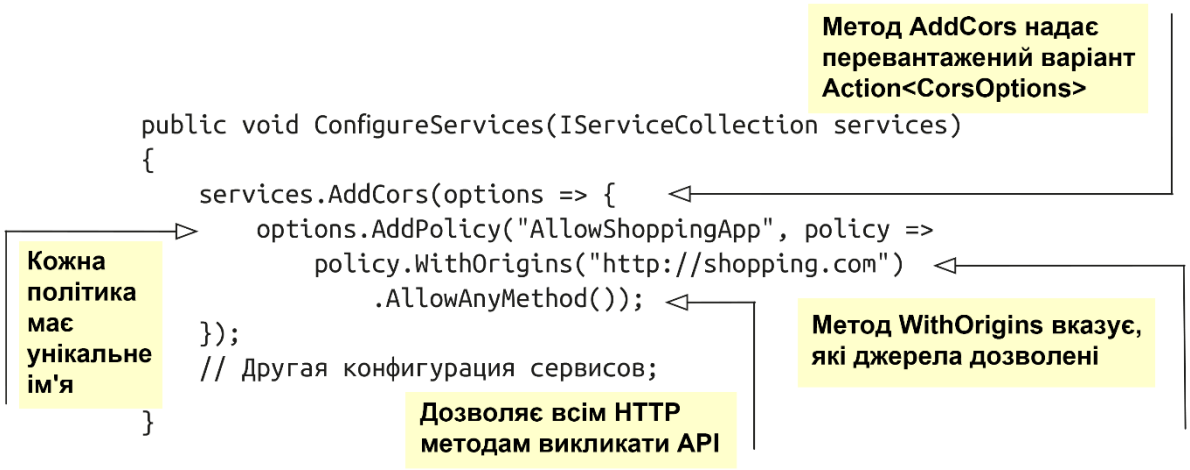


Рисунок 3.1 – Налаштування політики CORS, щоб дозволити запити з певного джерела

При перерахуванні джерел в методі WithOrigins() слід переконатися, що в них немає завершуючого символу у вигляді косої риси «/»; у протилежному випадку джерела не співпадуть й наші запити не будуть виконані.

Після того, як ми визначили політику CORS, ми можемо застосувати її до свого додатку. На рисунку 3.2 показано застосування політики "AllowShoppingApp" з усіма додатками за допомогою CorsMiddleware шляхом виклику методу UseCors() у методі налаштування файлу Startup.cs.

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddCors(options => {
    options.AddPolicy("AllowShoppingApp", policy =>
         policy.WithOrigins("http://shopping.com")
             .AllowAnyMethod());
});

var app = builder.Build();
app.UseRouting();
app.UseCors("AllowShoppingApp");
app.UseAuthentication();
app.UseAuthorization();

app.MapGet("/api/products", () => new string[] {});

app.Run();
```

Додає проміжне програмне забезпечення CORS та використовує AllowShoppingApp як політику за замовчуванням

Рисунок 3.2 – Додавання проміжного ПО CORS та налаштування політики CORS за замовчуванням

Як і в разі використання всіх компонентів проміжного ПО, важливим є порядок обробки запиту цим компонентом CORS. Необхідно розмістити виклик UseCors() після методу UseRouting() і до методу UseEndpoints(). Проміжне ПО для CORS має перехоплювати запити з різних джерел до вашого веб-API, щоб мати можливість генерувати правильні відповіді на попередні запити та додавати необхідні заголовки. Зазвичай проміжне ПО для CORS розміщується перед викликом UseAuthentication().

Маючи таке проміжне програмне забезпечення для API, додаток для онлайн торгівлі тепер може виконувати запити з різних джерел. Ми можемо викликати API з сайту http://shopping.com, і браузер пропускає CORS-запит, як показано на рисунку 3.3. Якщо ми зробимо той самий запит із домену, відмінного від http://shopping.com, запит залишиться заблокованим.

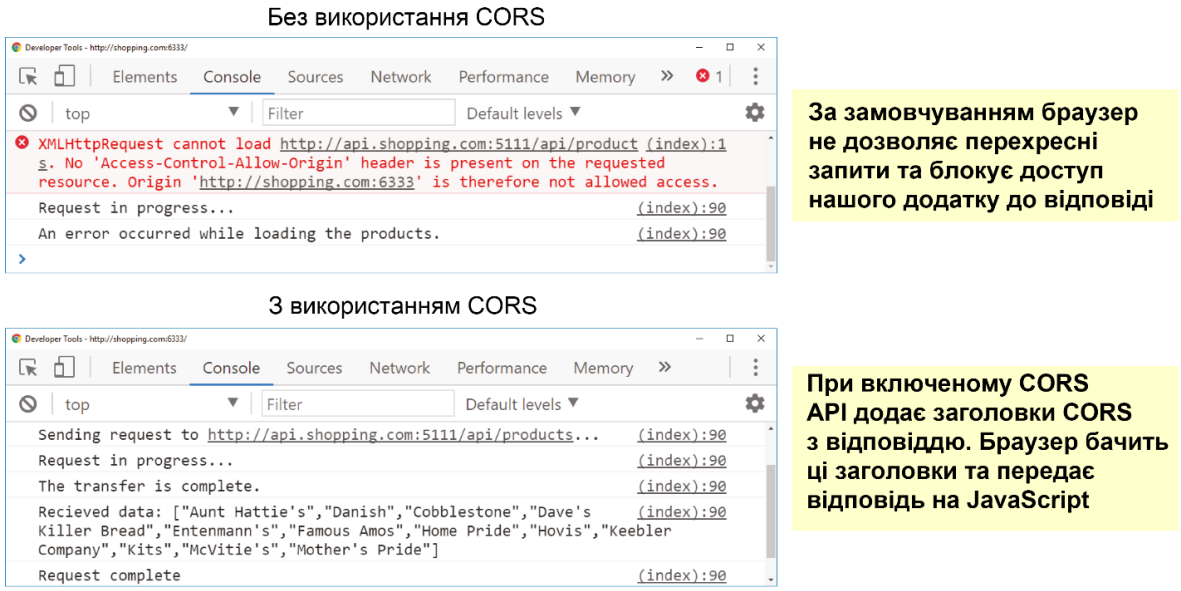


Рисунок 3.3 – Реалізація політики CORS

3.1.2 Додавання політики CORS до певних кінцевих точок

Глобальне застосування політики CORS до всього додатку може виявитися зайвим. Якщо в нашому API тільки частина кінцевих точок можуть обробляти запити з різних джерел, потрібно буде налаштовувати CORS лише для цих конкретних кінцевих точок.

ASP.NET Core має метод RequireCors(), який можна застосувати до окремих кінцевих точок або груп маршрутів у мінімальних API, а також атрибут [EnableCors], який дозволяє вибрати політику для застосування до даного контролера або методу дії.

Обидва ці методи додають метадані CORS в кінцеву точку, яка використовується CorsMiddleware для визначення застосовної політики. Ось чому CorsMiddleware слід розміщувати після RoutingMiddleware, щоб CorsMiddleware знав, яка кінцева точка була обрана та яку політику CORS застосувати.

За допомогою методу RequireCors() та атрибуту [EnableCors] ми можемо застосовувати різні політики CORS до різних кінцевих точок. Наприклад, ми можемо дозволити GETзапити до всього вашого API з домену http://shopping.com, але при цьому для інших типів HTTP запитів налаштувати

доступ на рівні окремих кінцевих точок, при цьому одночасно дозволяючи будь-якому джерелу отримати доступ до кінцевих точок списку продуктів .

Політики CORS задаються під час виклику методу AddCors(), викликаючи метод AddPolicy() та присвоюючи політиці ім'я. Якщо ми використовуємо політики, специфічні для кінцевої точки, замість виклику UseCors("AllowShoppingApp") слід додати проміжне програмне забезпечення без політики за умовчанням, викликавши тільки UseCors().

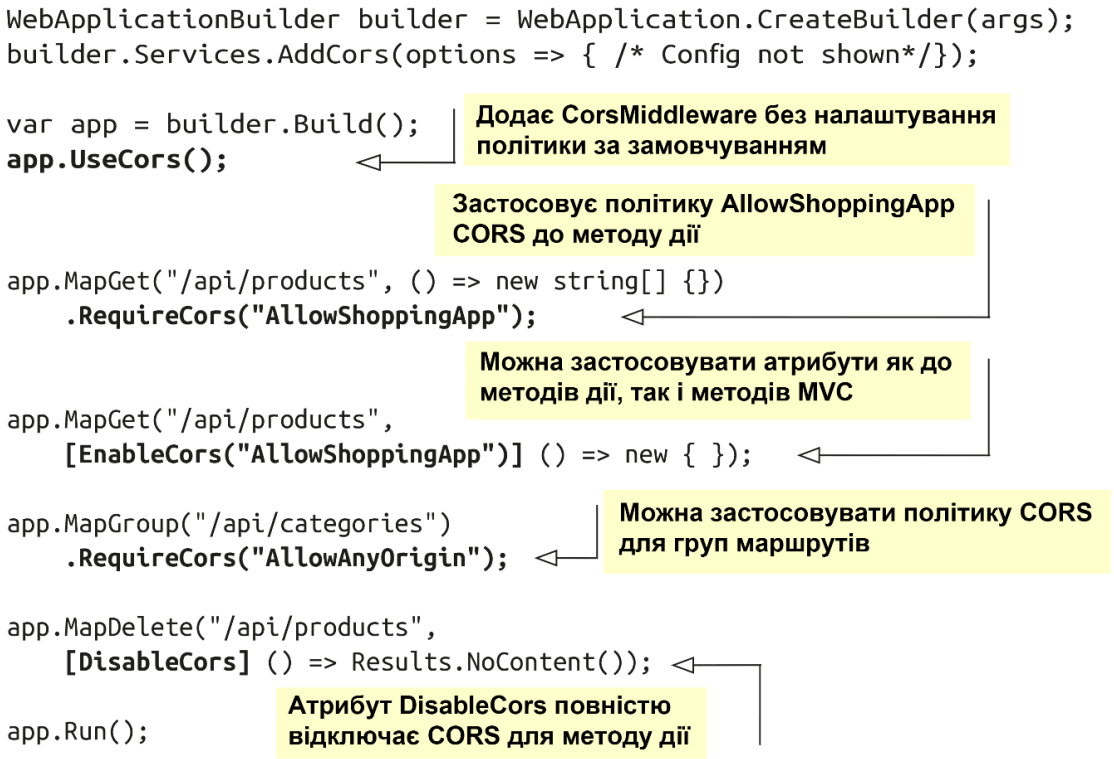


Рисунок 3.4 – Застосування політики CORS до кінцевих точок мінімального API

Якщо ми хочемо застосувати політику CORS до більшої частини методів дії, але хочемо використовувати іншу політику або повністю відключити CORS для деяких з них, то можемо передати політику за умовчанням при налаштуванні проміжного програмного забезпечення, наприклад, за допомогою UseCors("AllowShoppingApp"). Методи дії з атрибутом [EnableCors("OtherPolicy")] матимуть пріоритет OtherPolicy, а для методів дій з атрибутом [DisableCors] CORS взагалі не буде активовано. Незалежно від

того, чи хочемо ми використовувати одну політику CORS за замовчуванням або кілька, необхідно налаштувати політики CORS для своєї програми у методі ConfigureServices (розділ 2).

3.2 Запобігання атакам з використанням впровадження SQL-коду

Атаки шляхом впровадження SQL коду є однією з найнебезпечніших загроз для застосування. Зловмисники створюють простий шкідливий код, який вони відправляють у наш додаток у вигляді традиційного введення на основі форм або шляхом налаштування URL-адрес і рядків запиту для виконання довільного коду у нашій базі даних. Ця вразливість може надати доступ зловмисникам до всієї нашої бази даних, тому дуже важливо знаходити та усувати будь-які подібні вразливості у своїх додатках [34].

SQL має архітектурну проблему, характерну для багатьох мов запитів: дані та команди знаходяться в одному рядку. Якщо SQL-запит є результатом поєднання рядків, деякі з яких були введені користувачем, це може бути фактором ризику.

3.2.1 Використання підготовлених інструкцій

Впровадження SQL-коду працює, тому що зловмиснику вдається переключити контекст усередині SQL. Варто пам'ятати, що інструкція SQL – це рядок із командами та даними.

Екранування введення може бути перспективною стратегією. Коли ми поміщаємо дані користувача в SQL-запит, це також вважається вводом, тому це правило можна застосовувати. Проте виявляється, що правильно екранувати спеціальні символи SQL не так просто, тому що тут дуже багато варіантів залежно від контексту. Усередині рядків на думку приходять одинарні лапки (і заміна їх на \' або ', залежно від бази даних, що використовується).

Щодо інших спеціальних символів, то вони перераховані у табл. 3.1 й

деякі з них залежать від конкретної системи баз даних.

Таблиця 3.1 – Спеціальні символи SQL

Спеціальні символи	Функціональне призначення
( ) [ ]	Групування
_ %	Підстановочні знаки
;	Завершення запиту
-- # /*	Коментар

Варіантів кілька, тож тут легко помилитися. Чому б не дозволити комусь іншому зробити всю тяжку роботу? Виявляється, що всі відповідні баз даних для платформи .NET підтримують підготовлені інструкції (які ще називають параметризованими запитами). Тут SQL-запит відправляється до бази даних у два етапи. Спочатку створюється інструкція, яка може містити параметри. На другому етапі ці параметри набувають значень. Лише після цього інструкція виконується.

Це простий, але ефективний підхід, тому що він нейтралізує проблему SQL архітектури: тепер зрозуміло, де команда, а де дані. Надаючи значення цим заповнювачам, база даних знає, що подана інформація повинна розглядатися лише як дані. Насправді, так ми відключили використання небажаного SQL-коду. Ось як може виглядати запит із параметрами:

```
SELECT * FROM users WHERE email=@email AND password=@password
```

Зверніть увагу, що перед параметрами стоїть символ @, а лапки навколо параметрів відсутні (інакше імена параметрів використовувалися буквально, як рядки). Існують різні варіанти синтаксису для підготовлених інструкцій. Той, що ми використовували досі (з префіксом @), є найбільш поширеним і зручним, оскільки кожен параметр може мати унікальне ім'я, що запам'ятовується. Також для позначення параметрів можна використовувати

знак запитання (?), а потім надавати значення за розташуванням, а не по імені. Очевидно, що такий варіант більш схильний до помилок, оскільки розташування параметра буде змінюватися, як тільки десь перед ним буде доданий інший параметр.

Відомий виняток – СУБД Oracle, оскільки імена параметрів тут починаються з двокрапки, і інструкція SQL може виглядати так:

```
SELECT * FROM users WHERE email=:email AND password=:password
```

Протягом тривалого часу найпоширенішим способом надсилання SQL-запитів до бази даних із додатку ASP.NET була технологія ADO.NET (Advanced Data Objects для .NET). Цей підхід також працює з ASP.NET Core. Що стосується Microsoft SQL Server, пакет Microsoft.Data.SqlClient (www.nuget.org/packages/Microsoft.Data.SqlClient) містить постачальника даних. (Є аналогічні пакети й інших основних баз даних, та їх способи реалізації підготовлених інструкцій настільки схожі, що ми розглядатимемо їх). На рисунку 3.5. показано, як використовувати підготовлені інструкції з цим пакетом.

Нам потрібно створити об'єкт команди, передати йому SQL, а потім надати значення параметрів для запиту. Нарешті, необхідно виконати команду, щоб надіслати запит до бази даних і можливо отримати запитані записи. Тут ми використовували SQL Server, але об'єкти команд інших баз даних (наприклад, Oracle-Command від Oracle, MySqlCommand від MySQL, SQLiteCommand від SQLite або NpgsqlCommand з PostgreSQL), по суті, використовують той же синтаксис.

Якщо ми дійсно працюємо з SQL, то підготовлені інструкції – мабуть, найкращий засіб захисту від впровадження SQL-коду. Однак можна взагалі не писати SQL-код.

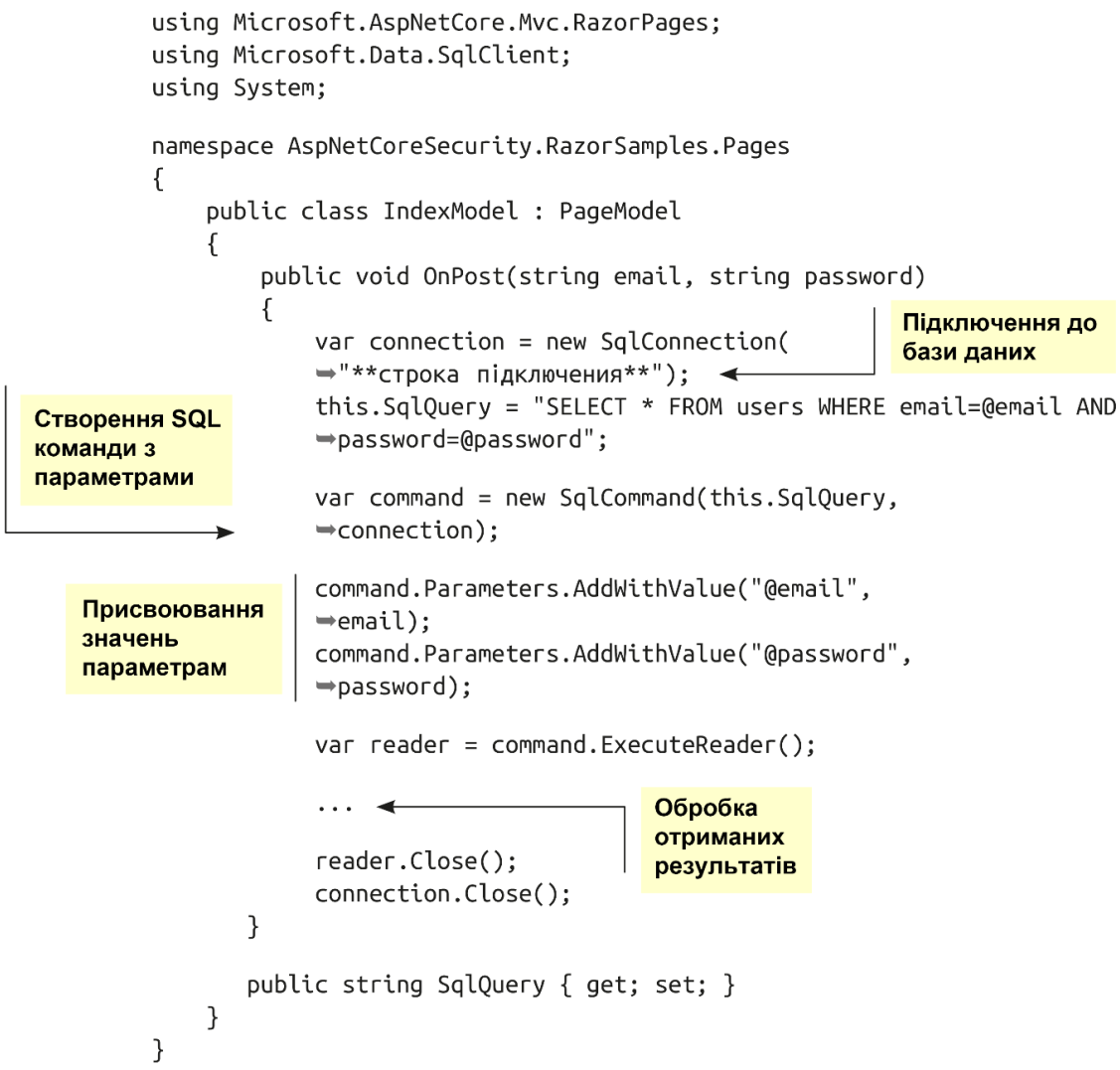


Рисунок 3.5 – Реалізація підготовленої інструкції при доступі до бази даних

3.2.2 Використання Entity Framework Core

Щоб позбавити розробників писати SQL-команди, компанія Microsoft пропонує інструмент Entity Framework (EF) Core. Це проста, кросплатформова версія популярної технології доступу до даних, яка дозволяє розробникам .NET працювати з базою даних за допомогою об'єктів .NET та усуває потребу у більшій частині коду для доступу до даних, який, зазвичай, доводиться писати [27]. У EF Core доступ до даних здійснюється за допомогою моделі. Модель складається з класів сутностей та об'єкта контексту, що представляє сеанс взаємодії з базою даних. Об'єкт контексту дозволяє виконувати запити та зберігати дані [28].

Псевдокод, який буде запитувати всіх користувачів з бази даних, щоб

знайти тих, хто має певну комбінацію адреси електронної пошти та пароля, має вигляд:

```
var users = db.Users.Where(user => user.email == email &&
                user.password == password)
```

Цей код робить майже те саме, що і SQL з попередніх прикладів, за винятком деяких особливих випадків, таких як бази даних з урахуванням регістру. Однак, тут немає конкатенації строк, а також відсутні параметри. Натомість є об'єкт, що позначає всіх користувачів в історії даних (db.Users) та синтаксис мови LINQ для вибору користувачів, які відповідають певним критеріям. У зловмисника немає можливості впровадити SQL, оскільки його немає. Ну, взагалі він є, але Entity Framework Core генерує його автоматично.

Але насправді це не зовсім повна картина. Entity Framework Core все ж таки дозволяє відправку SQL-запитів безпосередньо в базу даних. Це буде працювати аналогічно фрагменту коду, який ми щойно продемонстрували:

```
var users = db.Users.FromSqlRaw("SELECT * FROM users WHERE
email='villain@example.com' AND password='noclue'").ToList();
```

Entity Framework Core не перевіряє SQL, він просто перенаправляється до бази даних. Якщо ми використовуємо тут конкатенацію рядків і дані користувача, то програма раптово знову стає вразливою для впровадження SQL-коду. Зручно, що є підтримка параметризованих запитів, де використовується синтаксис, аналогічний синтаксису методу платформи .NET, String.Format(). Але варто переконатись, що ми використовуємо його лише безпосередньо у виклику FromSqlRaw (рисунок 3.6):

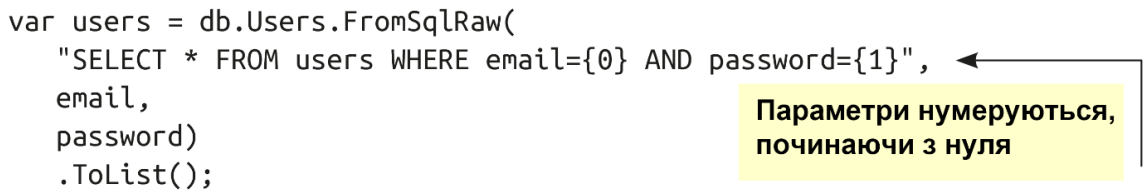


Рисунок 3.6 – Захист від провадження SQL-коду в Entity Framework Core шляхом використання параметризованого запиту

Ця інструкція належним чином екранує два значення для двох параметрів. В якості альтернативи можна використовувати інтерполяцію рядків (рисунок 3.7). Знову ж таки, два значення параметрів будуть правильно екрановані.

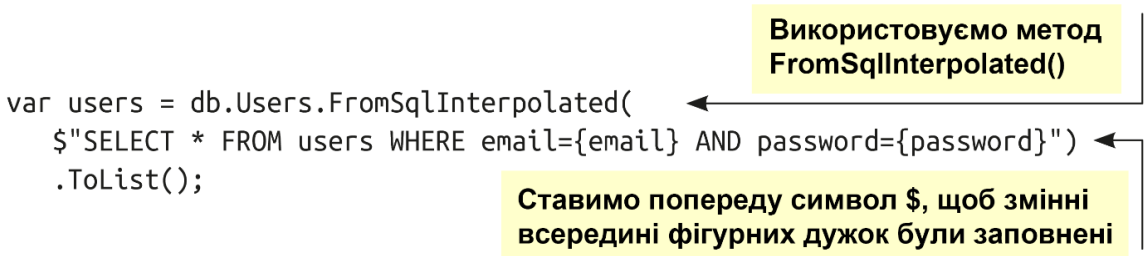


Рисунок 3.7 – Захист від провадження SQL-коду в Entity Framework Core шляхом використання інтерполяції рядків

Використання інструментів об'єктно-реляційного відображення, таких як Entity Framework Core, при всій своїй зручності, може створювати деякі проблеми. При їх застосуванні SQL-запити генеруються на основі наданої інформації (наприклад, об'єкта та його зв'язків). У деяких ситуаціях результат згубно впливає на продуктивність. Особливо це може статися, коли в одному запиті об'єднуються кілька таблиць. При роботі з таблицями без первинного ключа SQL може бути більш підходящим способом. Тому навіть найзатятіші прихильники Entity Framework Core завжди шукатимуть шляхи, в яких оригінальні SQL-запити є найкращим варіантом.

3.3 Система автентифікації та авторизації ASP.NET Core Identity

На багатьох сайтах користувачам надається можливість виконати вхід, а веб-сервер вирішує, які дії можна дозволити виконувати користувачу. При використанні одного постачальника входу в обліковий запис для різних сайтів (технологія єдиного входу) або при роботі з API ситуація ускладнюється. Фреймворк ASP.NET Core поставляється з вбудованою системою керування користувачами ASP.NET Core Identity, яка дозволяє користувачам реєструватися, виконувати вхід в обліковий запис і виходити з нього, реалізуючи при цьому різні протоколи й стандарти для захисту API й додатків.

3.3.1 Базові налаштування ASP.NET Core Identity

При створенні нового проекту в ASP.NET Core деякі механізми ASP.NET Core Identity вже вбудовані у додаток, якщо проект створено правильно. Найпростіший спосіб – використовувати Visual Studio. Створення нового веб-програми ASP.NET Core, незалежно від того, чи працюємо ми з Razor Pages або MVC, надає опцію Authentication Type (тип автентифікації) як частину майстра створення проекту (рисунок 3.8).

Якщо обрати варіант Individual Accounts (Індивідуальні облікові записи), додаток буде поставлятися з можливістю керування користувачами «з коробки». Після запуску додатку відкриється домашня сторінка (рисунок 3.9). Зверніть увагу на посилання Register (Реєстрація) та Login (Вхід) у правому верхньому кутку (рисунок 3.9) веб-додатку. Без додаткової конфігурації або коду можна реєструвати нових користувачів програми, а потім виконувати вхід, використовуючи ці облікові дані.

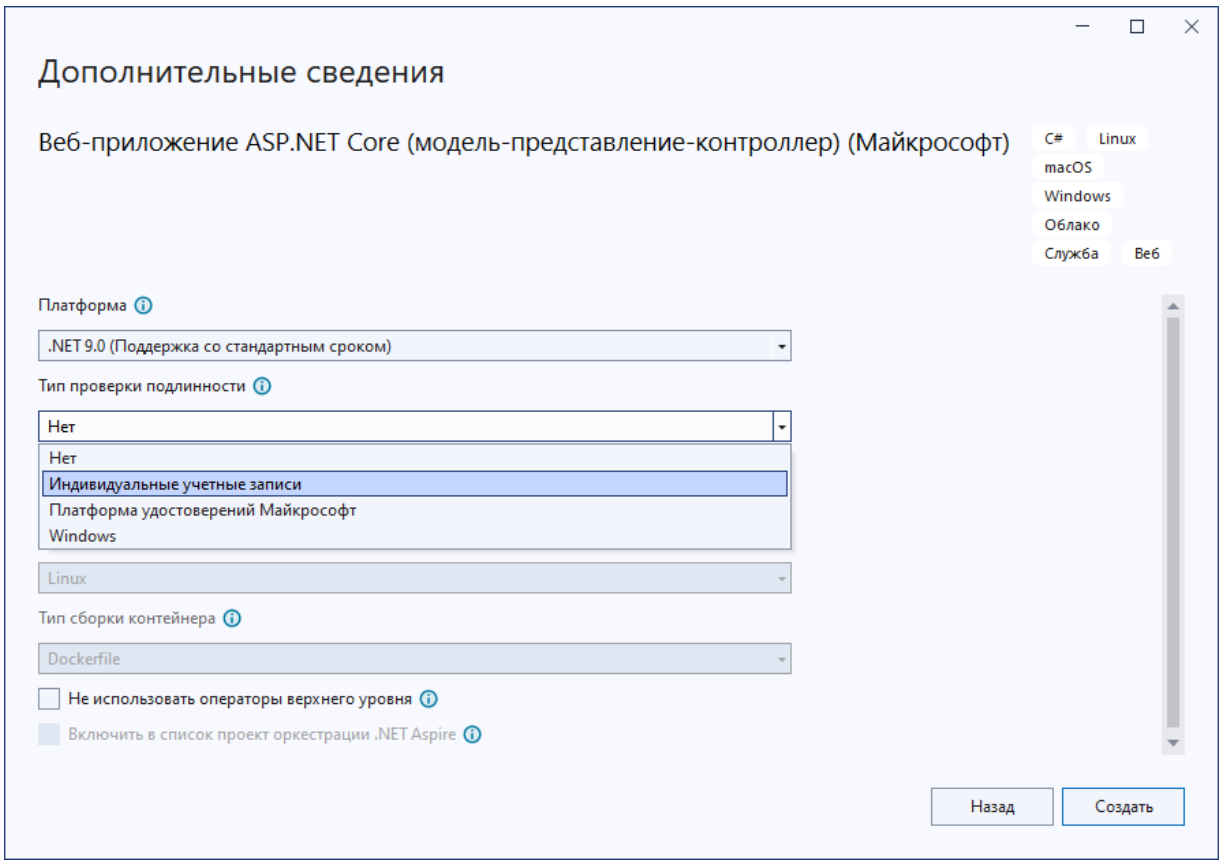


Рисунок 3.8 – Вибір типу автентифікації у майстрі створення проекту Visual Studio

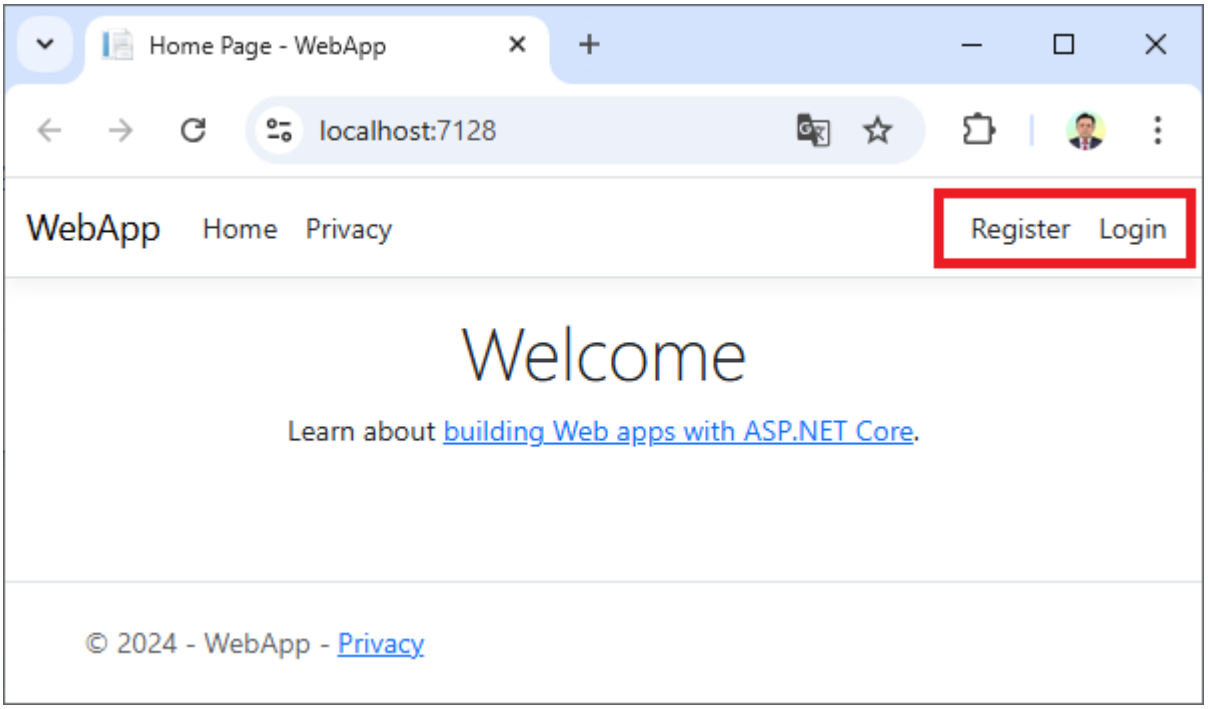


Рисунок 3.9 – Створений за допомогою шаблону веб-додаток

Наступним кроком є генерація середовищем коду тих сторінок, які ми хочемо створити (рисунок 3.10). Для цього потрібно клацнути правою кнопкою миші по проекту веб-додатку в браузері рішень Visual Studio та вибрати Add/New Scaffolded Item (Додати/Створити шаблонний елемент), Identity (Ідентичність) та зачекати кілька секунд, поки Visual Studio ініціалізує створення коду у фоновому режимі.

Також необхідно обрати клас DataContext. Краще задати той, який вже є частиною додатку, або створити новий контекст даних. Він використовується для того, щоб додаток знав, де зберігати та витягувати дані управління користувачами. Якщо ми вже маємо клас, який позначає користувача в додатку, ми також можемо вказати його в майстрі. Однак цей крок не є обов'язковим.

Через деякий час Visual Studio згенерує всі необхідні файли, і ми побачимо, що є реалізацією за замовчуванням (рисунок 3.11).

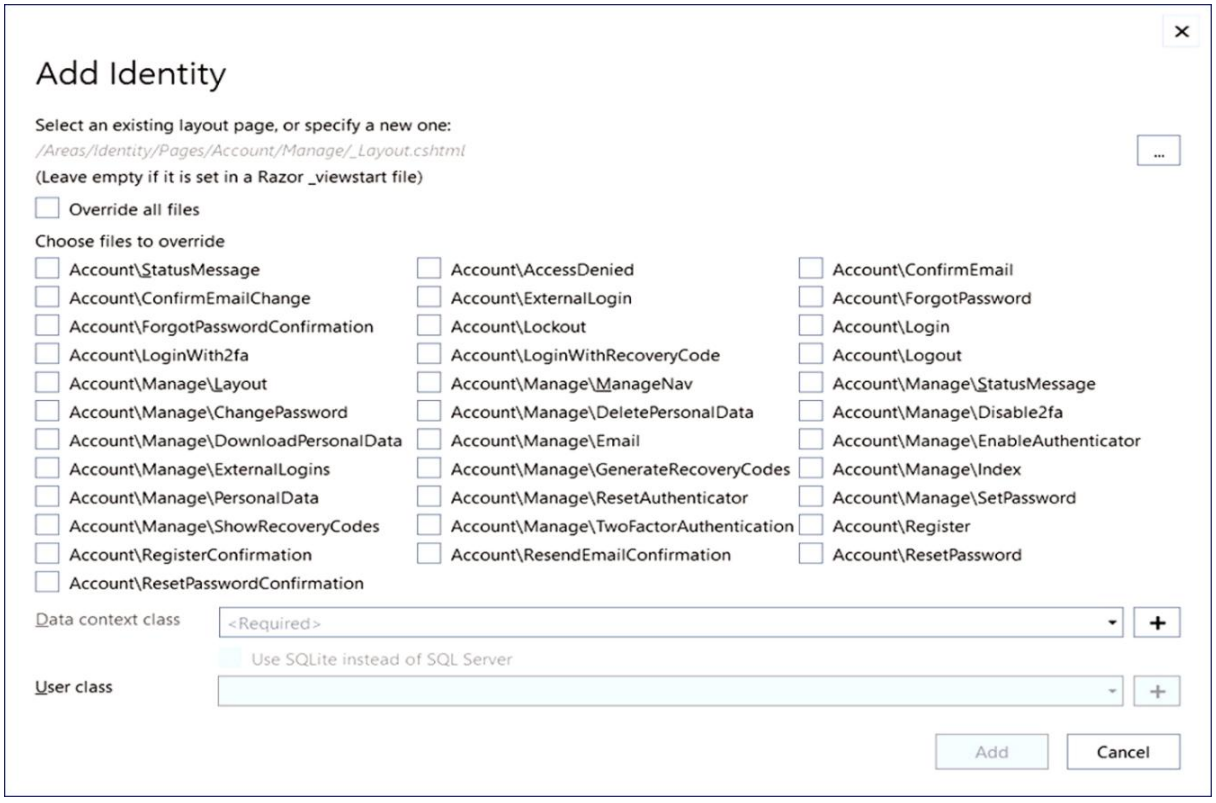


Рисунок 3.10 – Вибір сторінок, які потрібно створити в додатку

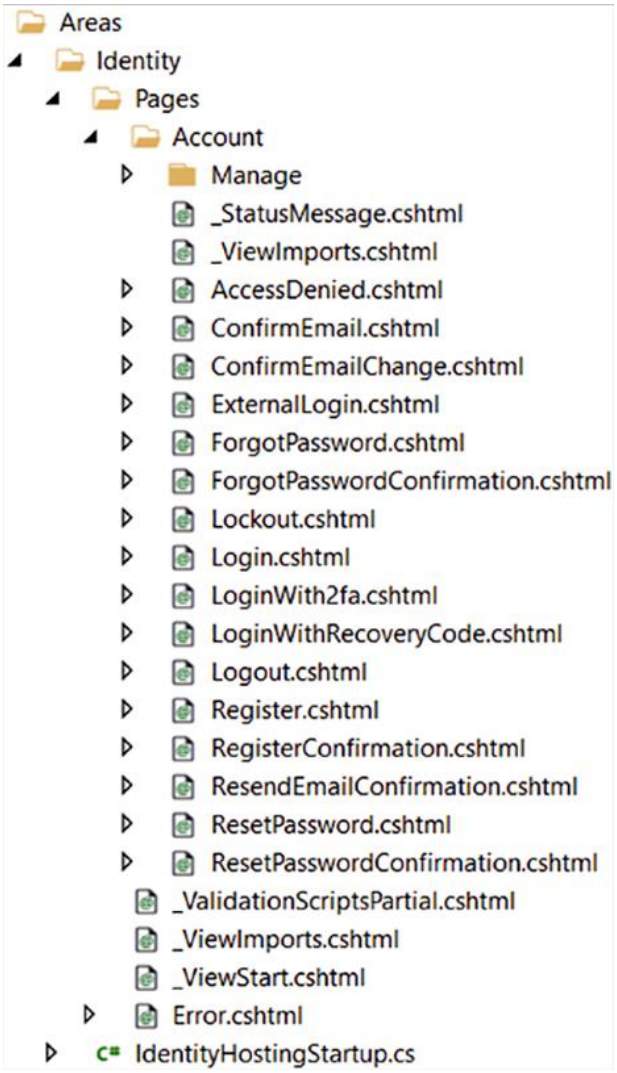


Рисунок 3.11 – Оглядач рішень після додавання вибраних сторінок

На рисунку 3.12 показано згенеровану схему бази даних, зокрема:

– у таблиці AspNetUsers містяться користувачі;

– у таблиці AspNetRoles містяться всі ролі користувачів, а в таблиці AspNetUserRoles показано, у яких користувачів які ролі. Роль – це категорія користувачів, які мають певні привілеї;

– для користувачів та ролей таблиці AspNetUserClaims та AspNet-RoleClaims містять пов'язані твердження. Твердження – це фрагмент інформації про користувача.

У базі даних немає стовбця для пароля, натомість зберігається хеш, що робить витік даних не таким катастрофічним.

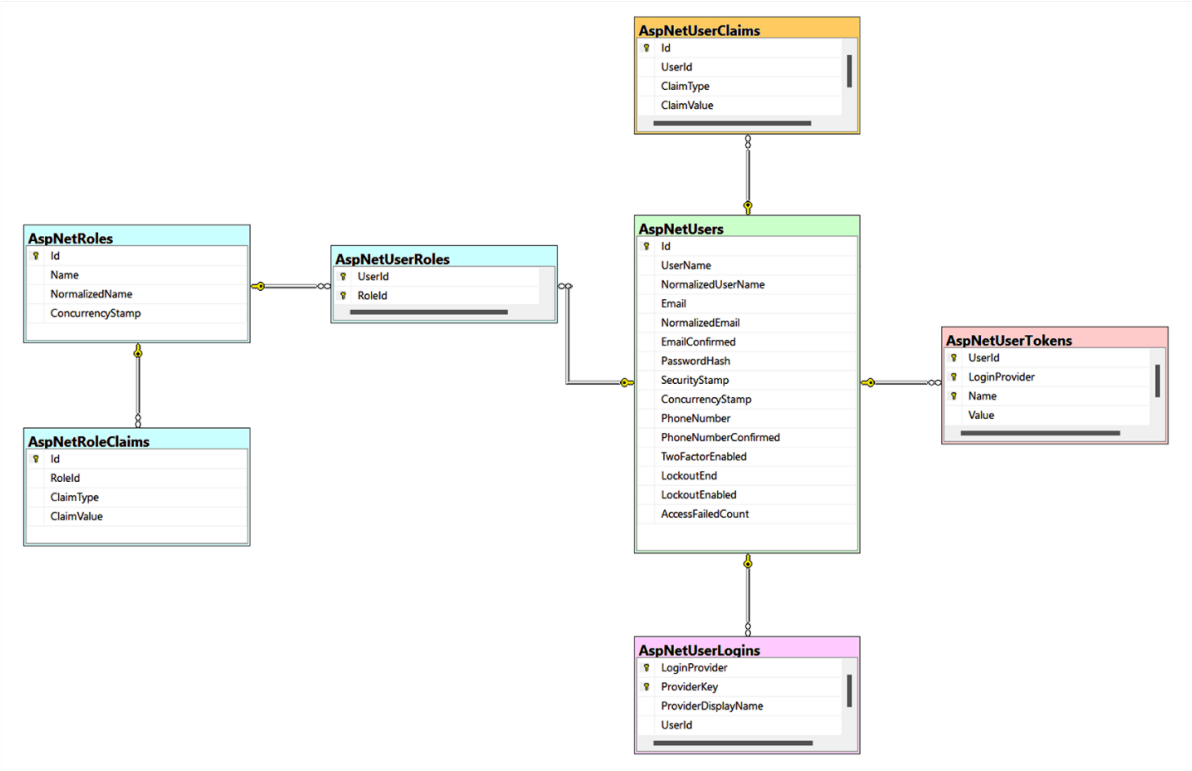


Рисунок 3.12 – Схема бази даних Identity

Ця структура бази даних чудово підходить для багатьох поширених ситуацій. Якщо необхідна додаткова функціональність, що не покривається вбудованою реалізацією, то її можна отримати, просто використовуючи доступні інтерфейси. Однак натомість ми розглянемо допоміжні вбудовані можливості, що вимагають додаткової конфігурації.

Надалі у роботі ми дослідимо, що може запропонувати даний шаблон. При розгляді будемо використовувати Razor Pages, хоча можна й використовувати шаблон MVC, оскільки базові концепції переважно однакові.

3.3.2 Розширення можливостей ASP.NET Core Identity

До цього моменту більша частина реалізації заходів безпеки створювалася ASP.NET Core Identity за нас автоматично, що насправді доволі непоганоо. Проте є багато способів, як налаштувати те, що у нас вже є, та вивчити і впровадити розширену функціональність.

3.3.2.1 Параметри пароля

При реєстрації нового користувача у веб-застосунку ASP.NET Core є ймовірність того, що перший пароль буде відхилений.

За замовчуванням, пароль повинен відповідати наступним критеріям (рисунок 3.13):

– його довжина не повинна бути меншою за шість символів;

– у ньому має бути хоча б одна мала літера;

– у ньому має бути хоча б одна велика літера;

– у ньому має бути хоча б одна цифра;

– він повинен містити хоча б один не буквенно-цифровий символ.

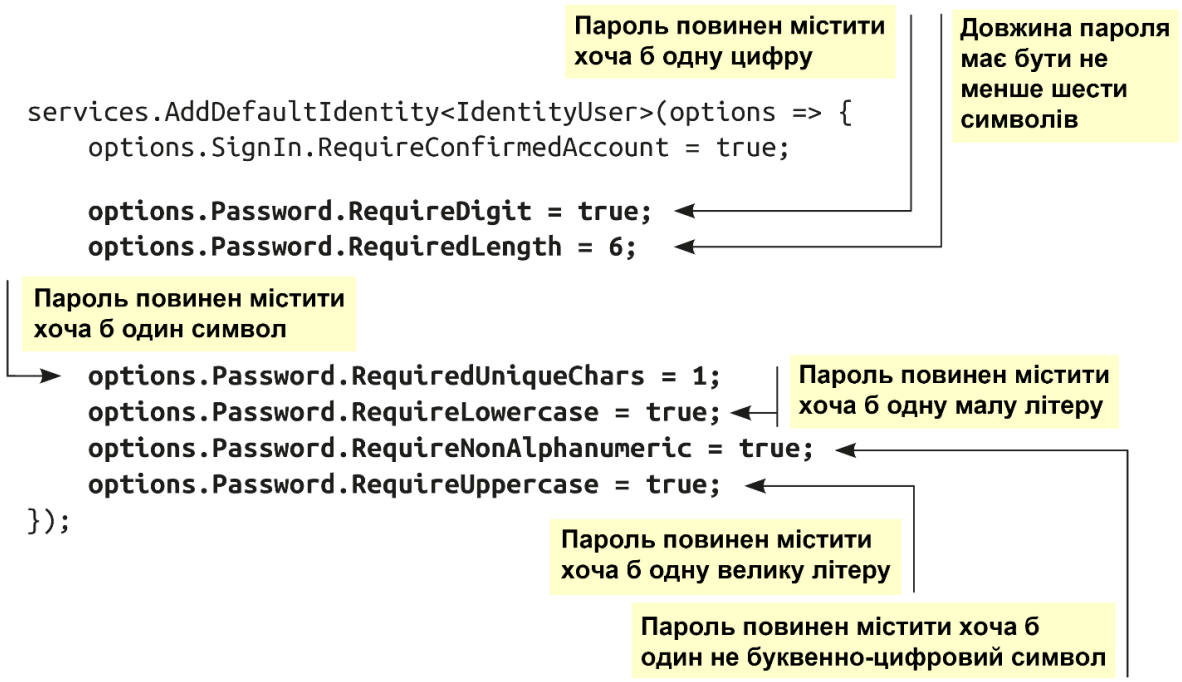


Рисунок 3.13 – Налаштування складності пароля по замовчуванню

Параметр RequiredUniqueChars (рисунок 3.13) використовується тільки для значень більше 1. Наприклад, якщо встановлено значення 2, то пароль повинен складатися з як мінімум двох різних символів.

З 2020 року Національний інститут стандартів і технологій США рекомендує, щоб довжина паролів становила не менше восьми символів і жодних вимог до складності (великі чи малі літери, цифри, спеціальні

символи) не пред'являє. Якщо ми хочемо застосувати це правило до наших додатків, то можемо виконати відповідні налаштування у файлі Program.cs за умови налаштування Identity.

Якщо ми хочемо, щоб мінімальна довжина пароля становила вісім символів, принаймні шість з яких повинні відрізнятися, і жодні інші обмеження застосовуватися не повинні, можна задати такі налаштування (рисунок 3.14):

```
services.AddDefaultIdentity<IdentityUser>(options => {
    options.SignIn.RequireConfirmedAccount = true;
    options.Password.RequireDigit = false;
    options.Password.RequiredLength = 8;
    options.Password.RequiredUniqueChars = 6;
    options.Password.RequireLowercase = false;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireUppercase = false;
});
```

Рисунок 3.14 – Налаштування власних вимог до пароля

Максимальна довжина пароля не регулюється. Оскільки паролі в будь-якому випадку мають бути хешовані, такого обмеження не повинно бути. Шаблон все ж таки обмежує довжину пароля до 100 символів. Цей параметр прихований у RegistrationModel (файл Registration.cshtml. cs) за допомогою атрибуту [StringLength].

3.3.2.2 Параметри файлів cookie

Після успішної автентифікації користувача автентифікаційний файл cookie отримує довге значення, яке неможливо вгадати. На рисунку 3.15 показано, як це може виглядати.

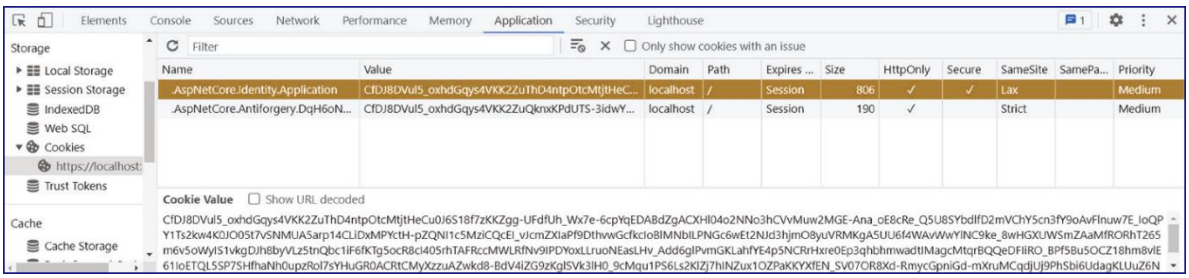


Рисунок 3.15 – Автентифікаційний файл cookie з параметрами по замовчуванню

Проаналізувавши рисунок 3.15, можна виявити такі параметри файлу cookie:

– ім'я файлу – AspNetCore.Identity.Application;

– прапори HttpOnly та Secure встановлені (останній лише при використанні протоколу HTTPS);

– режиму SameSite (забезпечення захисту від атак на міжсайтові підробки запиту CSRF) присвоєно значення Lax;

– термін дії файлу cookie закінчується при закритті браузера, тому дата закінчення терміну дії не вказана.

Усі ці та інші параметри можна змінити у файлі Program.cs, викликавши метод ConfigureApplicationCookie(), як це показано на рисунку 3.16.

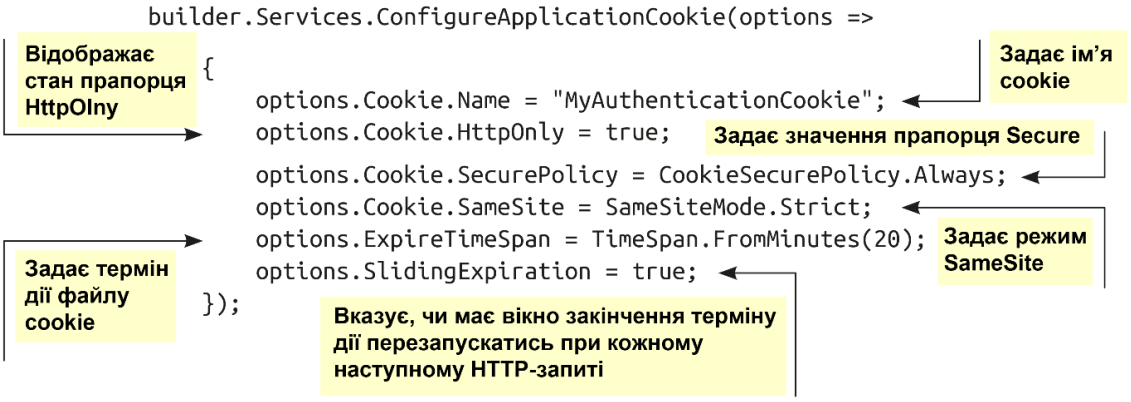


Рисунок 3.16 – Налаштування параметрів файлу cookie

Після виконання необхідних налаштувань файл cookie набуває вигляду:

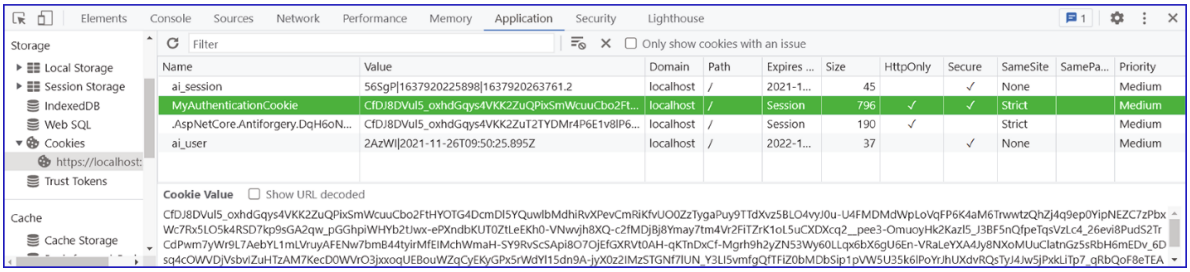


Рисунок 3.17 – Автентифікаційний файл cookie з оновленими параметрами

Всі оновлення конфігурації відображаються в cookie, за винятком дати закінчення терміну дії, яка зберігається в корисному навантаженні файлу cookie. Також було створено два нових файли cookie – ai_user та ai_session, що реалізують вікно закінчення терміну дії.

3.3.2.3 Блокування користувачів

Покарання користувачів за невдалі спроби введення пароля – міра з побічними ефектами. Тимчасове блокування користувачів після введення кількох неправильних паролів може запобігти їх пошуку шляхом повного перебору, але це також простий механізм, який використовується з метою не дати добропорядним користувачам отримати доступ до веб-додатку. Якщо зловмисник знає ім'я користувача своєї жертви, може легко заблокувати його обліковий запис, якщо додаток дозволяє це зробити.

Існує три параметри блокування, що підтримуються ASP.NET Core Identity:

– MaxFailedAccessAttempts – кількість невдалих спроб входу, перш ніж обліковий запис буде заблоковано (за замовчуванням – 5);

– DefaultLockoutTimeSpan – як довго користувачі не можуть отримати доступ до програми (за замовчуванням – 5 хв.);

– AllowedForNewUsers – визначає, чи можуть нові користувачі також бути заблоковані перед тим, як виконати вхід навіть один раз (за промовчанням – true).

Ці параметри застосовуються під час виклику методу

AddDefaultIdentity(). На рисунку 3.18. показано їх використання у фрагментів коду. Значення параметрів тут вказано по замовчуванню, але вони легко можуть бути замінені на бажані.

```
builder.Services.AddDefaultIdentity<IdentityUser>(options => {
    ...
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(5);
    options.Lockout.AllowedForNewUsers = true;
});
```

Рисунок 3.18 – Реалізація параметрів блокування користувачів

При виконанні входу в обліковий запис, використовуючи метод PasswordSignInAsync() із класу SignInManager, поведінку блокування можна перевизначити. Один із сценаріїв полягає в тому, що ми віддаємо перевагу зручності використання, а не безпеці, і не хочемо, щоб користувачі блокувалися, якщо хтось знає ім'я користувача та просто випадково пробує різні паролі. Створений шаблон робить це за допомогою наступного виклику у Login.cshtml.cs:

```
var result = await
_signInManager.PasswordSignInAsync(Input.Email,
Input.Password, Input.RememberMe, lockoutOnFailure: false);
```

3.3.2.4 Двофакторна автентифікація

Один із найбезпечніших підходів до запобігання крадіжці облікового запису – додати другий фактор при вході до облікового запису. Цей процес називається двофакторною автентифікацією (2FA). Другий фактор (першим є пароль) часто є одноразовим кодом або в текстовому повідомленні, або в одному з додатків-автентифікаторів (найчастіше використовуються програми від компаній Google і Microsoft). Якщо мобільний пристрій з такою програмою

перестає працювати, раніше згенеровані коди відновлення надають системі «чорний хід», що дозволяє увійти в обліковий запис і, наприклад, налаштувати інший пристрій.

Зручним є те, що створений додаток ASP.NET Core Identity вже підготовлений для двофакторної автентифікації, тому ми можемо подивитись його реалізацію та побачити, як це працює і що можна налаштувати. Щоб перейти у розділ двофакторної автентифікації в інтерфейсі користувача управління необхідно вказати шлях:

https://localhost:12345/Identity/Account/Manage/TwoFactorAuthentication,

Отриманий результат показано на рисунку 3.19.

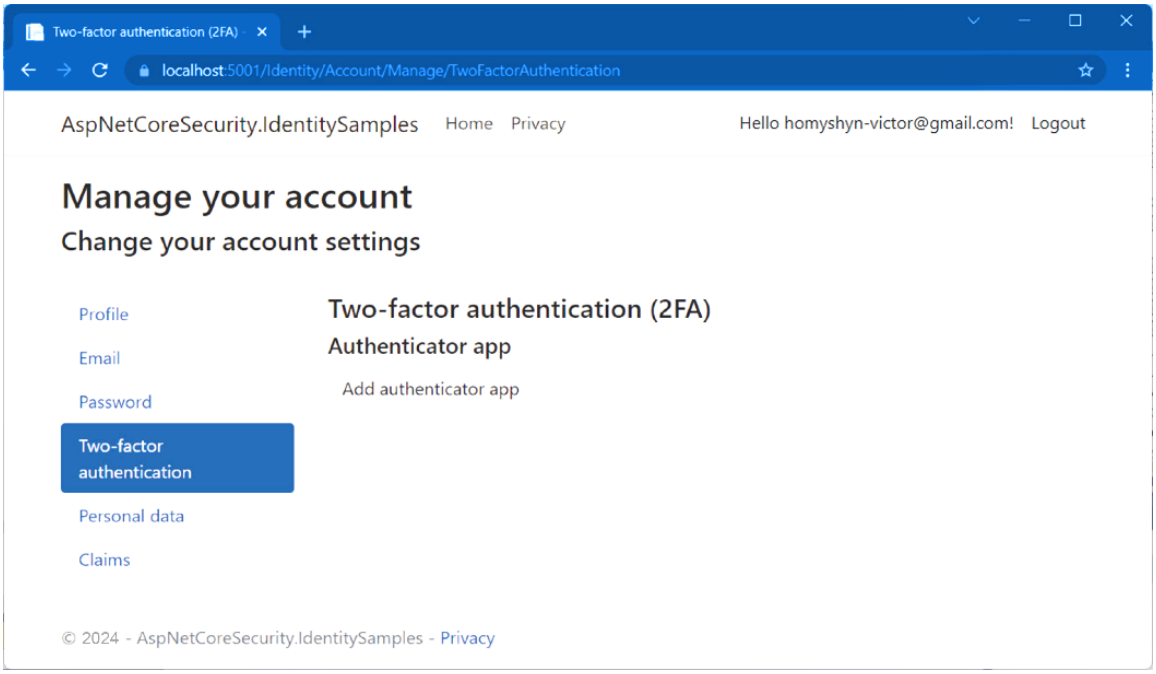


Рисунок 3.19 – Сторінка налаштувань двофакторної автентифікації

Для налаштування додатку-автентифікатора нам потрібно включити в проект бібліотеку генерації QR-коду, яка не входить до складу шаблонів. Команда розробників ASP.NET Core рекомендує використовувати QRCode.js автора Шима Сангміна. Її код доступний на сайті GitHub [29]. Достатньо просто завантажити один файл JavaScript й можна приступати до роботи без будь-яких вимог до керування пакетами.

ZIP-архів із бібліотекою генерування QR-коду завантажуємо із [30]. З цього архіву беремо файл qrcode.js та копіюємо його в наш проект у папку веб-додатку wwwroot > js (рисунок 3.20).

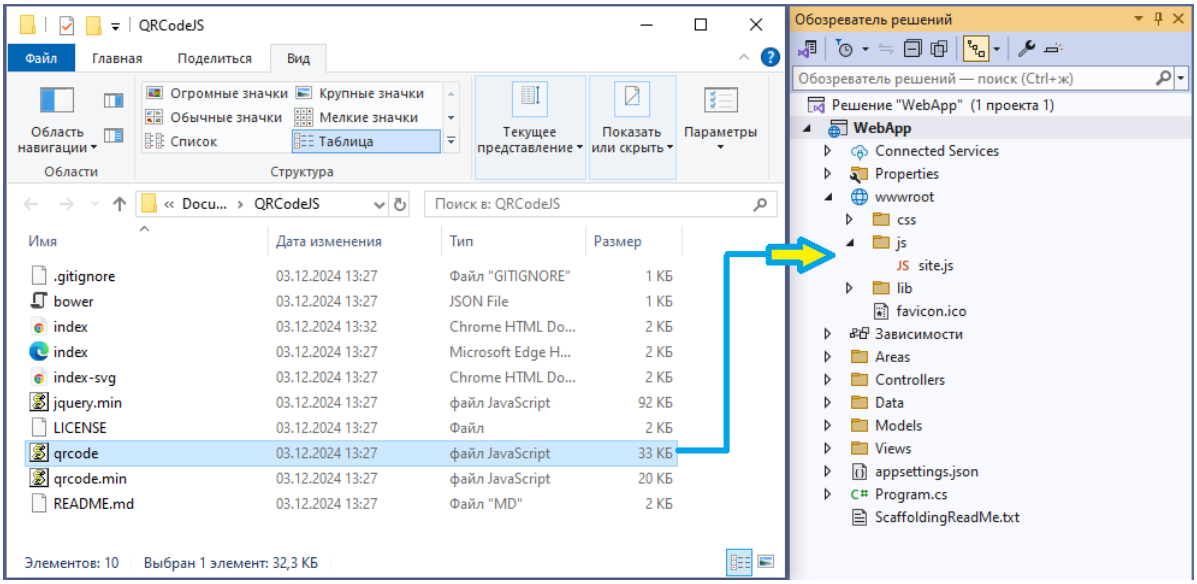


Рисунок 3.20 – Налаштування бібліотеки генерації QR-коду

У тій же папці створюємо файл enableAuthenticator.js наступного змісту:

```
$(function() {
    new QRCode(
        $("#qrCode")[0],
        {
            text: $("#qrCodeData").data("url"),
            width: 150,
            height: 150
        });
});
```

Створення нового QR-коду

Заповнювач, де буде відображатися QR-код

Отримання URL-адреси, яка закодована в QR-коді, з HTML-елемента qrCodeData

Рисунок 3.21 – Код JavaScript для створення QR-коду програми-автентифікатора

Цей фрагмент коду створює QR-код у заповнювачі на сторінці з ідентифікатором qrCodeData. Включаємо обидва файли до розділу Scripts (Сценарії) на сторінці EnableAuthenticator.cshtml (у папці Areas\Identity\Pages\

Account\Manage) (рисунок 3.22). Переконуємося, що спочатку завантажуємо бібліотеку генерування QR-коду, потім файл enableAuthenticator.js.

```
@section Scripts {
    <partial name="_ValidationScriptsPartial" />
    <script src="~/js/qrcode.js"></script>
    <script src="~/js/enableAuthenticator.js"></script>
}
```

Рисунок 3.22 – Додавання необхідних файлів в розділ сценаріїв

Після цього шукаємо наступну розмітку на сторінці:

```
<div class="alert alert-info">Learn how to
<a href="https://go.microsoft.com/fwlink/?Linkid=852423">
enable QR code generation</a>.</div>
```

Оскільки ми вже знаємо, як активувати генерування QR-коду, даний код видаляємо. Після оновлення сторінки маємо наступне (рисунок 3.23):

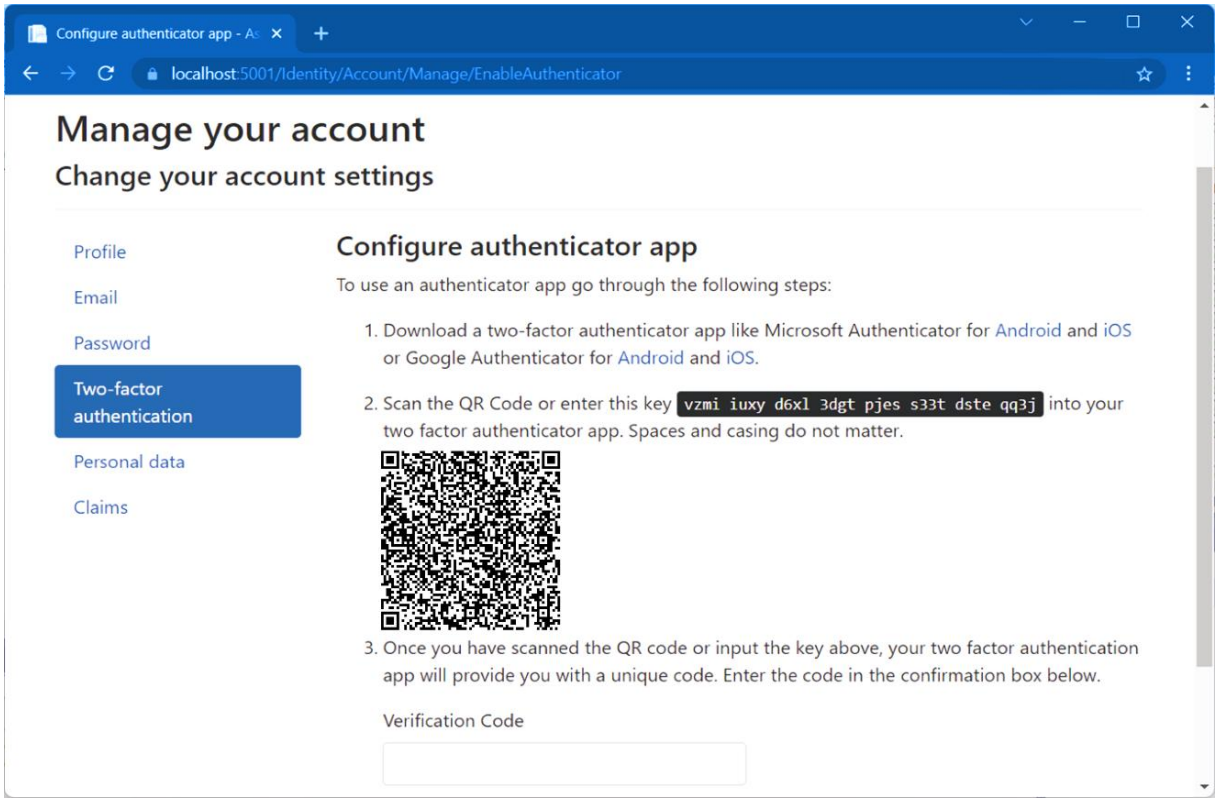


Рисунок 3.23 – Сторінка налаштувань з QR-кодом

Надалі використаємо програму-автентифікатор. Найпоширенішими є програми від компаній Microsoft та Google. Відскановуємо QR-код, після чого у нас з'являється запис у цій програмі і ми бачимо одноразовий пароль на основі часу, так званий TOTP, який змінюється кожні 60 с (рисунок 3.24).

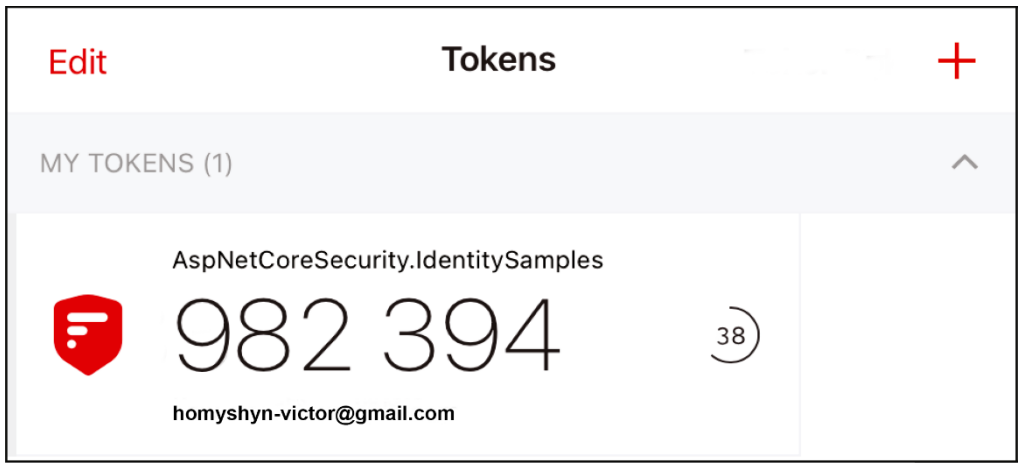


Рисунок 3.24 – Програма-автентифікатор показує одноразовий пароль на основі часу для веб-додатку

Вводимо поточний одноразовий пароль на основі часу в текстове поле на сторінці двофакторної автентифікації та натискаємо кнопку Verify (Перевірити). Якщо пароль вірний, то автентифікація налаштована правильно. Ми також отримали десять кодів відновлення, які дозволять нам виконати вхід навіть без пристрою з програмою автентифікатором, але використовувати кожен з цих кодів можна буде тільки один раз.

Наступного разу, коли ми спробуємо виконати вхід зі своїм ім'ям користувача та паролем, нам буде запропоновано ввести одноразовий код із програми-автентифікатора (рисунок 3.25). Також можна натиснути на посилання «You can log in with a recovery code», тобто можна увійти з кодом відновлення та «витратити» один із десяти кодів.

Ми розглянули текстові повідомлення як альтернативний механізм для другого фактора. Документація ASP.NET Core міститься детальний опис інструкцій для кількох SMS-сервісів [31].

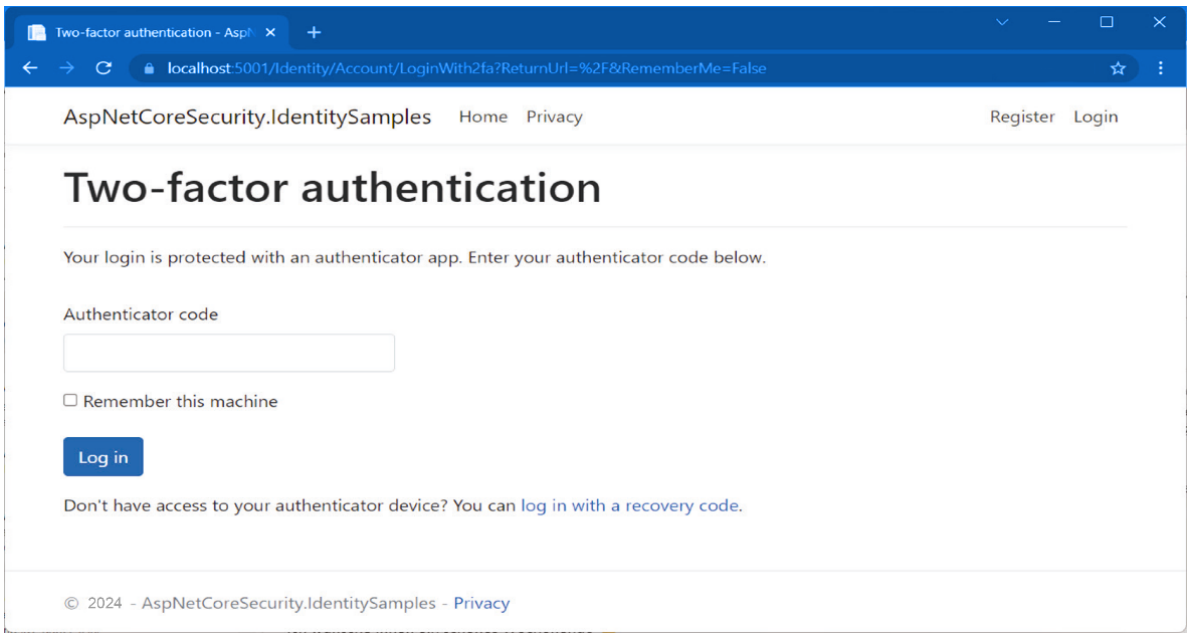


Рисунок 3.25 – Двохфакторна автентифікація вимагає одноразовий пароль на основі часу для входу

3.3.2.5 Автентифікація із зовнішніми постачальниками

Згідно з дослідженнями, середній користувач має не менше 10-20 паролів. Для багатьох користувачів вхід до стороннього додатку з використанням одного з існуючих облікових записів у Google, Facebook, Twitter, Apple чи інших ресурсів – приваблива пропозиція. Для забезпечення даної функціональності переважно використовується OAuth та OpenID Connect. Тим не менше, оскільки ми вже маємо створений додаток, ASP.NET Core Identity підтримує сторонні сервіси автентифікації. Розглянемо цю можливість детальніше.

Деталі реалізації різняться залежно від постачальника, але зазвичай послідовність дій наступна:

1. Встановити пакет NuGet для конкретного постачальника автентифікації.

2. Зареєструвати додаток у постачальника, який повинен надати нам облікові дані додатку (ідентифікатор та секретний токен).

3. Зберегти ці облікові дані в ASP.NET Core, щоб встановлений пакет NuGet та ASP.NET Core Identity виконували свою справу.

Наприклад, ми використовуємо облікові записи Microsoft для автентифікації. Додаток у хмарі Microsoft Azure служитиме сполучною ланкою між нашим додатком та обліковими записами Microsoft. Варто пам'ятати, що встановлення цього додатка може виявитися затратним. Перейдемо в розділ App Registrations (Реєстрація додатків) в Azure або за допомогою панелі пошуку у верхній частині, або за прямим посиланням (може виглядати по іншому): https://portal.azure.com/#blade/Microsoft_AAD_RegisteredApps/ApplicationsListBlade. Перейдемо за посиланням New Registration (Нова реєстрація) та зареєструємо новий додаток. На рисунку 3.26 показано форму, в яку можна ввести відомості про додаток.

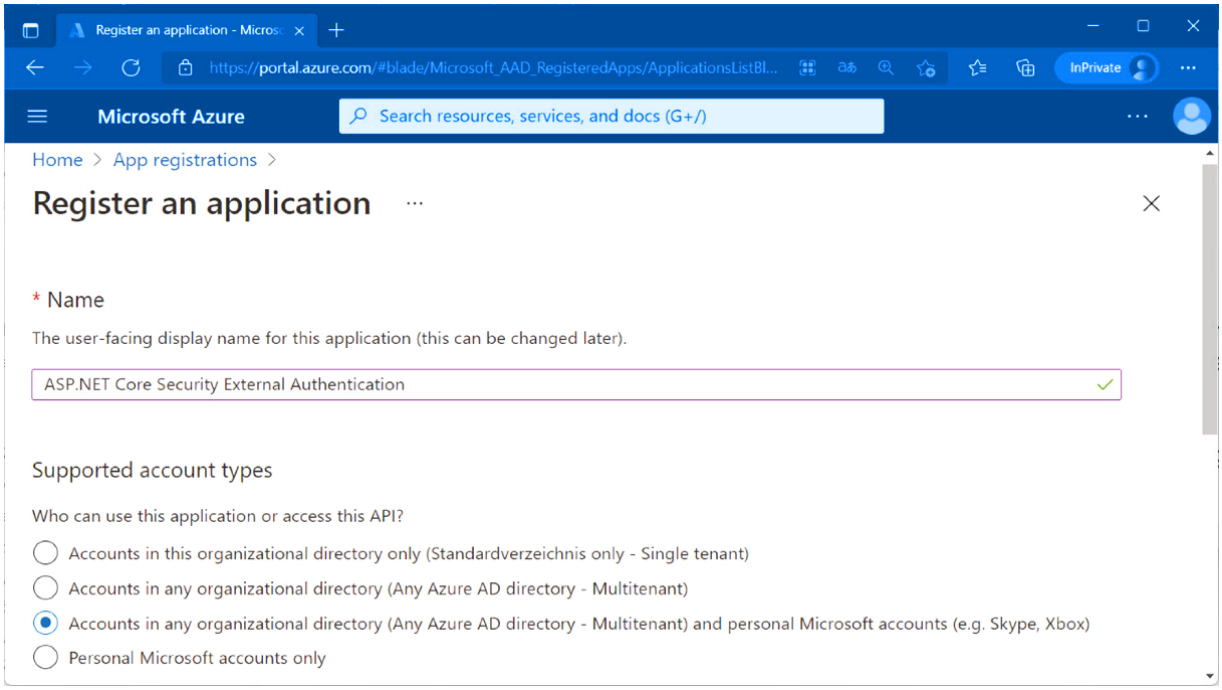


Рисунок 3.26 – Реєстрація додатку в Microsoft Azure

Вказуємо ім'я програми та обираємо параметр «Accounts in any organizational directory (Any Azure AD directory – Multitenant) and personal Microsoft accounts (e.g., Skype, Xbox)), тобто облікові записи в будь-якому організаційному каталозі (будь-який каталог Azure AD – мультиорендний) та особисті облікові записи Microsoft (наприклад, Skype, Xbox). У розділі Redirect URI (URI перенаправлення) вводимо URL-адресу, що складається з основної

частини та /signin-microsoft. У прикладі це https://localhost:5001/signin-microsoft, але переконуємося, що ми використовуємо адресу свого сервера. Кінцевої точки signin-microsoft ще немає, пакет NuGet зареєструє її пізніше.

Клацаємо по кнопці Register (Реєстрація) і на наступній сторінці запишимо показаний ідентифікатор програми (довгий ідентифікатор GUID). Потім знайдемо посилання «Certificates & Secrets» (Сертифікати та секрети) на панелі навігації зліва. Клацаємо кнопкою миші по цьому елементу та створюємо новий секрет клієнта (потрібно лише заповнити опис). Після цього ми побачимо вікно, схоже на рисунку 3.27. Секретне значення (з позначкою Value (Значення)) є актуальним для наступних кроків, а не GUID з позначкою Secret ID.

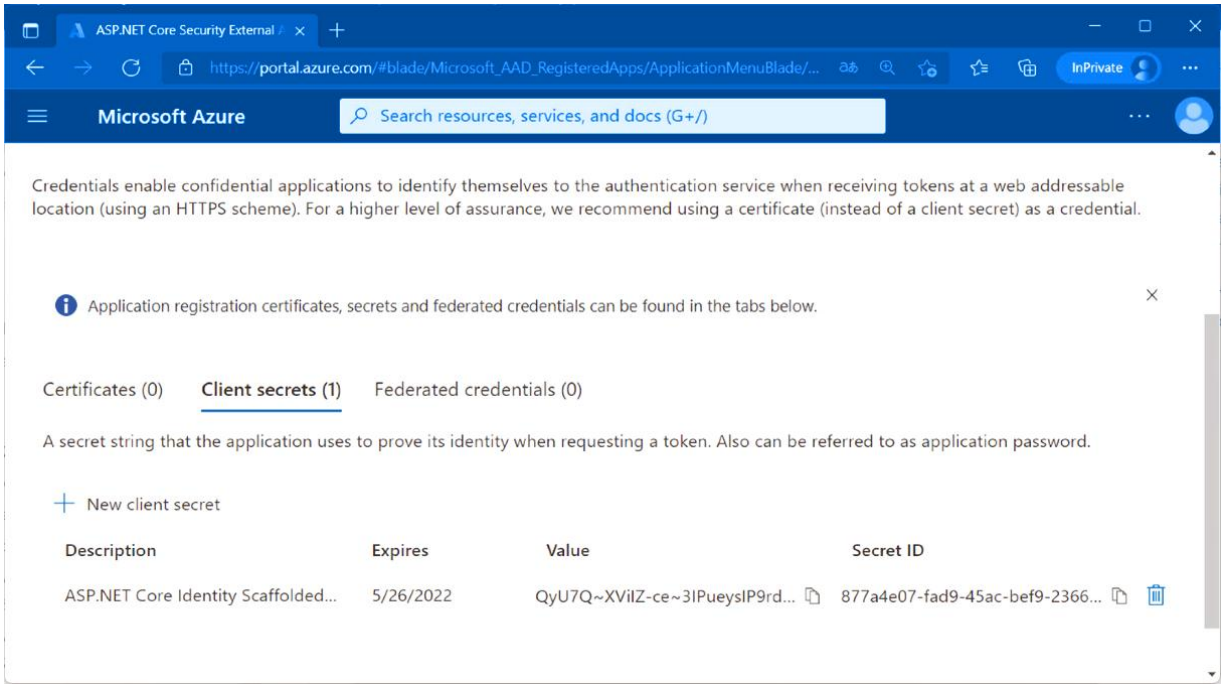


Рисунок 3.27 – Створення секрету клієнта

Повернувшись у веб-додаток, додамо його ідентифікатор та секретне значення з Azure до конфігураційного файлу appsettings.json (рисунок 3.28). Структура та імена для цих параметрів конфігурації довільні, якщо ми будемо використовувати цей самий підхід при читанні цієї інформації пізніше:

```
{
  "MicrosoftAuthentication": {
    "AppID": "212ff228-effd-4111-8118-878ade97abad",
    "Secret": "QyU7Q~XViIZ-ce~3IPueysIP9rdZJ_zUjiem~"
  },
  ...
}
```

Рисунок 3.28 – Запис в конфігураційному файлі appsettings.json

Тепер настав час встановити пакет NuGet, що взаємодіє з додатком Azure. Це пакет Microsoft.AspNetCore.Authentication.MicrosoftAccount. Ми використали версію, що відповідає нашій версії .NET.

У файлі Program.cs знайдемо виклик методу AddAuthentication(), а за його відсутності, додаємо його. Пакет NuGet визначив спосіб розширення AddMicrosoftAccount(). Прописуємо параметри конфігурації з файлу apsettings.json наступним чином (рисунок 3.29):

```
builder.Services.AddAuthentication()
    .AddMicrosoftAccount(options =>
    {
        options.ClientId =
     builder.Configuration["MicrosoftAuthentication:AppID"];
        options.ClientSecret =
     builder.Configuration["MicrosoftAuthentication:Secret"];
    });
```

Рисунок 3.29 – Налаштування файлу Program.cs

Тепер коли ми знову запустимо додаток й спробуємо зареєструвати обліковий запис побачимо, що з'явилася опція реєстрації через Microsoft (рисунок 3.30).

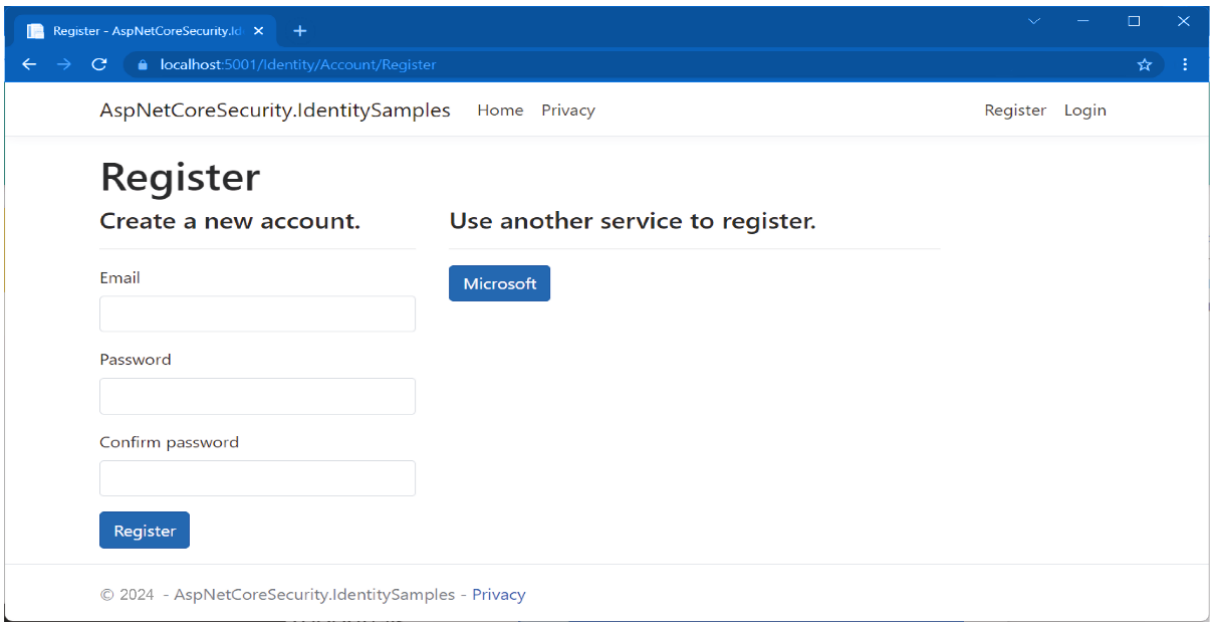


Рисунок 3.30 – Користувач може виконати вхід з обліковим записом Microsoft

Якщо ми виберемо цей варіант, будемо перенаправлені на екран входу до облікового запису Microsoft. Після авторизації необхідно дати свою згоду на отримання додатком нашої особистої інформації (зазвичай, це адреса електронної пошти) (рисунок 3.31).

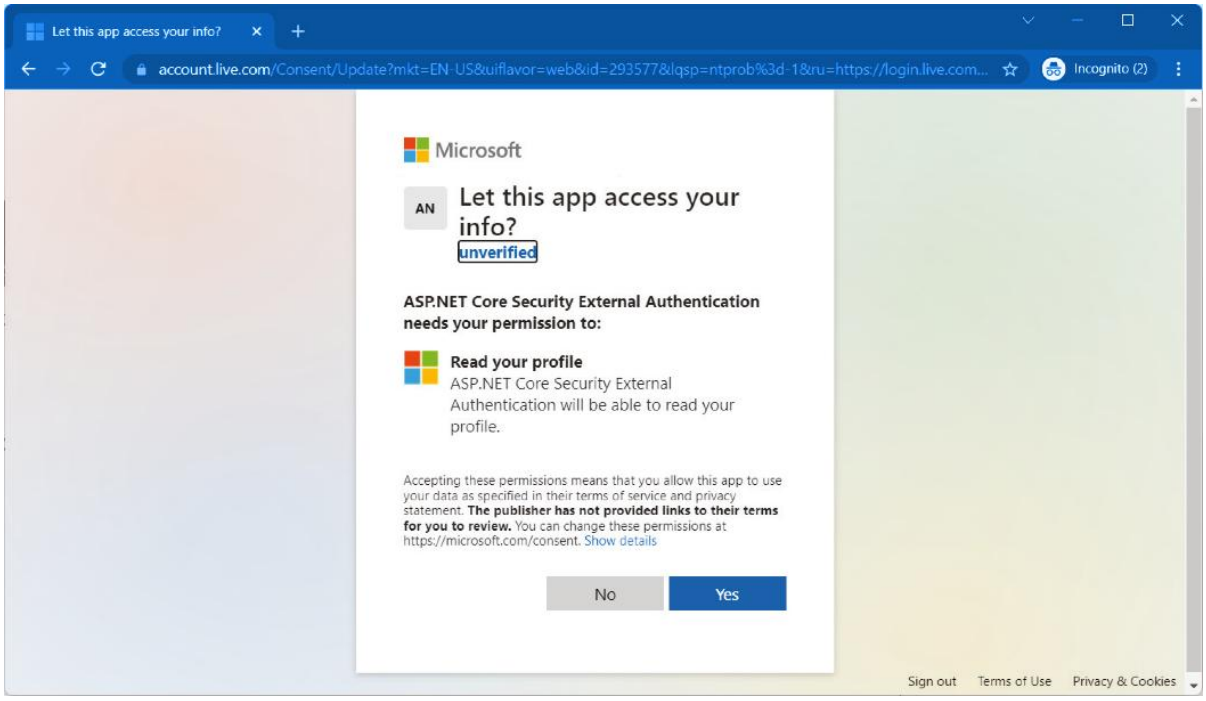


Рисунок 3.31 – Запит на отримання веб-додаткам доступу до даних облікового запису Microsoft

Як тільки ми погодимося, то будемо перенаправлені назад у програму, де нам може знадобитися підтвердити адресу електронної пошти, після чого ми зможемо використовувати програму з обліковим записом Microsoft. На майбутнє ми можемо увійти до веб-додатку, використовуючи цей запис.

З точки зору коду, зусилля з реалізації тут досить невеликі, оскільки більшу частину роботи виконує пакет NuGet. У самому додатку перший крок – це виклик методу GetExternalAuthenticationSchemesAsync() класу SignInManager, який повертає всі зовнішні автентифікаційні сервіси, зареєстровані в додатку. Сторінка входу в обліковий запис використовує це, щоб вирішити, які зовнішні автентифікаційні посилання показувати. У нашому випадку – це відображення посилання з написом «Microsoft». Натиснувши на це посилання, ми надсилаємо HTTP-запит з методом POST на сторінку ExternalLogin.cshtml. Тут запускається перенаправлення на сторінку входу у сторонньому сервісі автентифікації (рисунок 3.32).

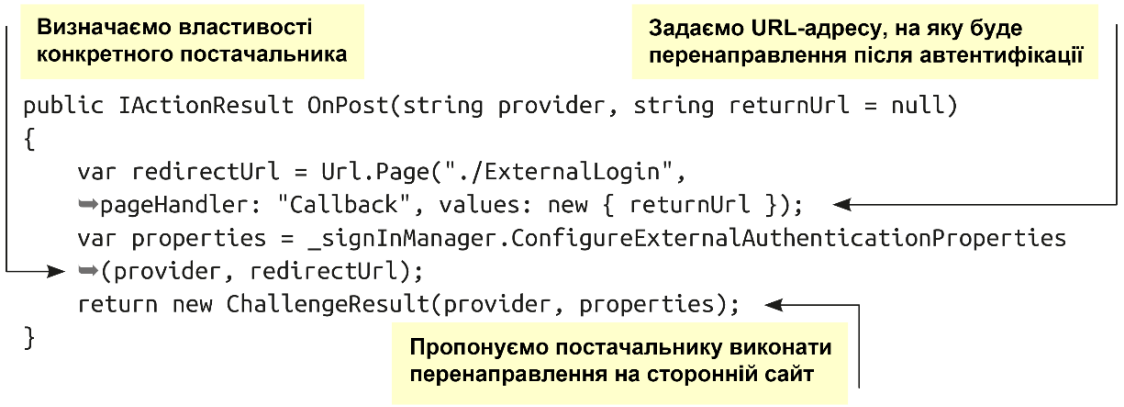


Рисунок 3.32 – Реалізація перенаправлення на сторонній сервіс автентифікації

У методі Callback, який спрацьовує, коли сторонній сайт автентифікації перенаправляє нас назад у додаток, центральний рядок коду – знову виклик SignInManager (рисунок 3.33).

```
var result = await
     _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
     info.ProviderKey, isPersistent: false, bypassTwoFactor : true);
```

Рисунок 3.33 – Лістинг коду для аналіз даних, що надходять від сервісу автентифікації

Метод ExternalLoginSignInAsync() аналізує дані, що надходять від сервісу автентифікації, а потім вирішує, був вхід успішним чи ні. Якщо користувач реєструється вперше, використовуючи зовнішній обліковий запис, то спочатку він повинен зареєструвати локальний обліковий запис (який потім буде пов'язаний із зовнішнім записом). В іншому випадку користувач отримає негайний доступ до програми.

Це лише один із більш ніж десятки можливих сервісів автентифікації, але незалежно від того, який з них ми використовуємо, модель постачальника ASP.NET Core Identity не потребує серйозних змін у коді, щоб усе це працювало, якщо є пакет NuGet. На ресурсі [31] можна знайти багато інших варіантів.

Варто також згадати і про проект aspnet-contrib, який містить набагато більше пакетів NuGet для зовнішньої автентифікації, які вже готові до використання у ASP.NET Core. Необхідно зайти на сторінку [32] та ввести у рядку пошуку owners:aspnet-contrib title:OAuth. На момент написання цієї роботи було показано понад 108 варіантів.

Висновки до розділу 3

1. CORS використовує HTTP заголовки для обміну даними з браузерами та визначає, які джерела можуть викликати наш API. У ASP.NET Core можна визначити кілька політик, які можуть застосовуватися або глобально до всього додатку, або до конкретних контролерів та дій. Ми можемо додати проміжне програмне забезпечення CORS, викликавши метод Use-Cors() класу WebApplication. Також можна застосувати CORS до методу дії або контролера веб-API, додавши атрибут [EnableCors] і вказавши ім'я політики, що застосовується.

2. Атаки з використанням впровадження SQL коду є найбільш поширеними при створенні SQL запитів вручну. Для їх запобігання необхідно використовувати параметризовані запити або фреймворк наприклад Entity Framework Core, який є менш вразливим для впровадження SQL коду.

3. Використання у веб-додатках ASP.NET Core Identity дозволяє пропрацювати всі аспекти керування користувачами та входом. При цьому ми можемо реалізувати власні варіанти, розширивши можливості даного API.

4. Атрибут [Authorize] можна використовувати для запобігання доступу користувачів, що не пройшли автентифікацію, до сторінок веб-додатку.

5. Доступ до ресурсів також може бути обмежений певними ролями, зокрема за допомогою атрибуту [Authorize]. Після настроюваної кількості невдалих входів обліковий запис може бути заблокований на певний час.

6. ASP.NET Core Identity підтримує двофакторну автентифікацію (2FA). Дещо доповнивши функціонал, додаток може генерувати QR-коди, що полегшують підключення додатків-автентифікаторів.

7. За допомогою спеціальних пакетів NuGet користувачі також можуть реєструватися у веб-додатку та проходити автентифікацію, використовуючи сторонній обліковий запис з таких ресурсів, як Facebook, Google або Microsoft.

ВИСНОВКИ

У кваліфікаційній роботі розв’язано актуальну задачу підвищення ефективності захисту веб-додатків, розроблених на базі фреймворку ASP.NET Core. При цьому отримано наступні результати:

1. Проведено детальний аналіз предметної області, що включає ознайомлення з особливостями та можливостями фреймворку ASP.NET Core, типами створюваних шаблонів додатків, зокрема: MVC, Razor Pages, Web API, Blazor. Описано компоненти та загрози типового веб-додатку, серед яких: клієнтська частина, сервер, база даних, зовнішні ресурси, шляхи взаємодії між ними. Зазначено, що феймворк ASP.NET Core має вбудовані засоби захисту. Частина з них активована за замовчуванням, інша – готова до налаштування й активації, для решти – потрібно писати власний код.

2. Розглянуто методи захисту ASP.NET Core веб-додатків, зокрема: захист від міжсайтового скриптингу, захист від міжсайтової підробки запиту, створення унікальних токенів з API для захисту даних, безпека використання ресурсів з різних джерел, виявлення та запобігання атакам з відкритим перенаправленням. По кожній з приведених загроз подано детальні кроки та код для додатків, які унеможливлюють експлуатацію даних вразливостей.

3. Запропоновано для запобігання атакам з використанням впровадження SQL коду використовувати компоненту Entity Framework Core у поєднанні з параметризованими запитами.

4. Розкрито можливості та переваги а також дано чіткі рекомендації щодо використання політики CORS, яка може застосовуватися або глобально до всього додатку, або до конкретних контролерів та дій.

5. Розширено базові можливості компоненти ASP.NET Core Identity, що дозволяє використовувати всі параметри керування користувачами та їх входом, двофакторну автентифікацію з можливістю генерування додатком QR-кодів та можливістю реєстрування й проходження автентифікації шляхом використання сторонніх облікових записів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бондаренко О., Ушкаленко І. Безпека web-додатків: Актуальні проблеми та їх аналіз. Формування ринкової економіки в Україні. 2017. № 38. С. 28–36.

2. Безпека веб-додатків та хакерські атаки / О. О. Боскін та ін. Вісник Херсонського національного технічного університету. 2023. № 3(86). С. 83–92. URL: https://doi.org/10.35546/kntu2078-4481.2023.3.11 (дата звернення: 15.11.2024).

3. Web Applications vulnerabilities and threats: statistics for 2019. Positive Technologies. URL: https://global.ptsecurity.com/analytics/web-vulnerabilities-2020 (дата звернення: 15.11.2024).

4. Ревнюк О., Постолюк А. Дослідження застосування адаптивних методик оцінювання ризиків безпеки для веб-додатків. Computer systems and information technologies. 2024. № 3. С. 34–43. URL: https://doi.org/10.31891/csit-2024-3-5 (дата звернення: 15.11.2024).

5. Опорний конспект лекцій з курсу «Безпека WEB ресурсів» для студентів спеціальності 125 «Кібербезпека та захист інформації». Тернопіль: ЗУНУ, 2023. 50 с. URL: http://dspace.wunu.edu.ua/bitstream/316497/49113/1/Безпека%20веб-ресурсів.pdf (дата звернення: 15.11.2024).

6. Microsoft выпустила .NET 9 с тысячами улучшений. Хабр. URL: https://habr.com/ru/news/858550/ (дата звернення: 15.11.2024).

7. Lock A. ASP. NET Core in Action, Third Edition. Manning Publications Co. LLC, 2023. 984 с.

8. Strauss D. Creating ASP.NET Core Web Applications. Berkeley, CA : Apress, 2021. 291 с. URL: https://doi.org/10.1007/978-1-4842-6828-5 (дата звернення: 15.11.2024).

9. Marcotte C. Architecting ASP.NET Core Applications: An atypical design patterns guide for .NET 8, C# 12, and beyond, Third Edition. Packt Publishing, 2024. 806 p.

10. Smith S. Architecting Modern Web Applications with ASP.NET Core and Azure. Redmond, Washington: Microsoft Corporation, 2023. 109 с.

11. Mezei R. A. Introduction to the Development of Web Applications Using ASP .Net (Core) MVC. Cham: Springer Nature Switzerland, 2024. URL: https://doi.org/10.1007/978-3-031-30626-6 (дата звернення: 15.11.2024).

12. Коноваленко І.В. Платформа .NET та мова програмування C# 8.0: навчальний посібник / Коноваленко І.В., Марущак П.О. Тернопіль: ФОП Паляниця В.А., 2020. 320 с.

13. Mark J Price. C# 13 and .NET 9 - Modern Cross-Platform Development Fundamentals - Ninth Edition: Start building websites and services with ASP.NET Core 9, Blazor, and EF Core 9. Packt Publishing, 2024. 828 p.

14. Brind M. ASP.NET Core Razor Pages in Action. Manning Publications Co. LLC, 2022. 456 p.

15. Freeman A. Pro ASP.NET Core MVC. Berkeley, CA : Apress, 2016. 1018 p. URL: https://doi.org/10.1007/978-1-4842-0397-2 (дата звернення: 15.11.2024).

16. Best Practices to Secure ASP.NET Core MVC Web Applications | Syncfusion Blogs. Syncfusion. URL: https://www.syncfusion.com/blogs/post/10-practices-secure-asp-net-core-mvc-app (дата звернення: 15.11.2024).

17. Sainty C. Blazor in Action. Manning Publications Co. LLC, 2022. 344 p.

18. Sanctis V. D. ASP. NET Core Web API. Manning Publications Co. LLC, 2023.

19. Wenz C. ASP.NET Core Security. Manning Publications Co. LLC, 2022. 368 с.

20. Norberg S. Advanced ASP.NET Core 8 Security. Berkeley, CA : Apress, 2024. 455 с. URL: https://doi.org/10.1007/979-8-8688-0494-6 (дата звернення: 15.11.2024).

21. Freeman A. Pro ASP.NET Core Identity: Under the Hood with Authentication and Authorization in ASP.NET Core 5 and 6 Applications. Berkeley, CA : Apress, 2021. 725 p. URL: https://doi.org/10.1007/978-1-4842-6858-2 (дата звернення: 15.11.2024).

22. Cross Site Scripting (XSS) | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-community/attacks/xss/ (дата звернення: 15.11.2024).

23. Nagarjun P., Shakeel S. Cross-site Scripting Research: A Review. International Journal of Advanced Computer Science and Applications. 2020. Т. 11, № 4. URL: https://doi.org/10.14569/ijacsa.2020.0110481 (дата звернення: 15.11.2024).

24. Work with SameSite cookies in ASP.NET Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/uk-ua/aspnet/core/security/samesite?view=aspnetcore-9.0 (дата звернення: 15.11.2024).

25. Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/uk-ua/aspnet/core/security/anti-request-forgery?view=aspnetcore-9.0 (дата звернення: 15.11.2024).

26. Hossain M. Chapter 1. The Core of CORS · CORS in Action: Creating and consuming cross-origin APIs. CORS in Action: Creating and consuming cross-origin APIs. URL: https://livebook.manning.com/book/cors-in-action/chapter-1/ (дата звернення: 15.11.2024).

27. Обзор Entity Framework Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/ru-ru/ef/core/ (дата звернення: 15.11.2024).

28. Smith J. Entity Framework Core in Action, Second Edition. Manning Publications Co. LLC, 2021. 624 p.

29. GitHub - davidshimjs/qrcodejs: Cross-browser QRCode generator for javascript. GitHub. URL: https://github.com/davidshimjs/qrcodejs (дата звернення: 15.11.2024).

30. JavaScript library for making QRCode: David Shim. URL: http://mng.bz/z4z1 (дата звернення: 15.11.2024).

31. Two-factor authentication with SMS in ASP.NET Core. Microsoft Learn: Build skills that open doors in your career. URL: https://learn.microsoft.com/en-

us/aspnet/core/security/authentication/2fa?view=aspnetcore-1.1 (дата звернення: 15.11.2024).

32. NuGet Gallery | Packages matching owners:aspnet-contrib title:OAuth. NuGet Gallery | Home. URL: https://www.nuget.org/packages?q=owners:aspnet-contrib+title:OAuth (дата звернення: 15.11.2024).

33. Деркач М.В., Хомишин В.Г., Гудзенко В.О. Тестування безпеки вебресурсу на базі інструментів для сканування та виявлення вразливостей. Наукові вісті Далівського університету. Електронне видання. 2023. № 25. DOI: https://doi.org/10.33216/2222-3428-2023-25-1.

34. Хомишин В. Г. Захист ASP.NET Core веб-додатків від впровадження SQL-коду. Актуальні задачі сучасних технологій : Матеріали XIII Міжнар. науково-техн. конф. молодих уч. та студентів, м. Тернопіль, 11–12 груд. 2024 р. Тернопіль, 2024.

ДОДАТОК А

Копії публікацій

Наукові вісті Далівського університету

№25 2023 рік.

РЕКОМЕНДОВАНО ДО ВИПУСКУ ВЧЕНОЮ РАДОЮ

СХІДНОУКРАЇНСЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

(ПРОТОКОЛ № 5 ВІД 28.12.2023 р.).

DOI: https://doi.org/10.33216/2222-3428-2023-25

ЗМІСТ/CONTENTS

DOI: https://doi.org/10.33216/2222-3428-2023-25-1
УДК: 004.056

Деркач М.В., Хомишин В.Г., Гудзенко В.О.

ТЕСТУВАННЯ БЕЗПЕКИ ВЕБРЕСУРСУ НА БАЗІ ІНСТРУМЕНТІВ ДЛЯ СКАНУВАННЯ ТА ВИЯВЛЕННЯ ВРАЗЛИВОСТЕЙ

У статті розглянуте актуальне питання тестування безпеки вебресурсу структурного підрозділу, створеного на базі системи керування вмістом WordPress, різними інструментами сканування та виявлення вразливостей, а саме Mozilla Observatory, Qualys, ImmuniWeb. Оскільки важливою умовою для безперервного процесу забезпечення безпеки бізнес-процесів сайту компанії, збереження ділової репутації, економічного зростання та розвитку бізнесу є саме регулярні діагностичні та відновлювальні процедури різного характеру та рівня у складі аудиту безпеки сайту, спрямованого на підвищення безпеки та надійності інтернет-ресурсу. Частина таких робіт присвячена пошуку вразливостей безпосередньо у структурі, виявлення помилок у коді та програмному забезпеченні сервера, якими зловмисники можуть атакувати та зламати сайт. В результаті проведено тестування безпеки вебресурсу структурного підрозділу зазначеними автоматизованими інструментами, кожний з яких виявив певні недоліки. Результат сканування двома останніми інструментами відповідає вищій категорії оцінки поточного рівня захищеності сайту. Завдяки інструменту Mozilla Observatory отримана посередня оцінка, що обумовлено більш широкими можливостями цього сервісу. Оскільки він застосовує інтегровані інструменти та рекомендації від OWASP, Probely, а також ті самі Qualys й ImmuniWeb, тим самим комплексна оцінка враховує не лише захист зашифрованого мережевого з'єднання з іншою системою за допомогою протоколу SSL/TLS. В цілому вебресурс забезпечує надійний обмін інформацією між браузером користувача та сервером. А з розвитком мережевих технологій та інтернету взаємодія різних систем, сервісів і додатків одна з одною набула значної актуальності, тому до тестування взаємодії варто підходити з усією серйозністю.

Ключові слова: *безпека, вебресурс, вразливість, криптографічні алгоритми, сканування, сертифікат, тестування.*

Вступ. Головне для власника вебсайту – це розуміння важливості безпеки власного сайту, розуміння необхідності забезпечувати його безпеку та перевіряти її (періодично), для чого і проводиться тестування безпеки. Тестування безпеки - це стратегія тестування, яка використовується для перевірки безпеки вебресурсу, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту додатків, хакерів, вірусів, несанкціонованого доступу до конфіденційних даних. Проведення тестування дозволяє виявити ці слабкі місця та усунути їх, забезпечуючи більш високий рівень безпеки для вебресурсу та його користувачів. На сьогодні це стає мірою необхідності, так як вебресурси постійно піддаються різноманітним загрозам, а вразливості можуть бути використані зловмисниками для атак або порушення принципів безпеки вебресурсу, таких як конфіденційність, цілісність та довіра, пошкодження та відновлення, доступність. Запобіжним заходом, який дозволяє отримати об'єктивну оцінку рівня захисту вебресурсу компанії, детальну інформацію про знайдені вразливості, можливі сценарії атак та рекомендації щодо їх усунення, стає аудит безпеки сайту, що являє собою низку процедур, спрямованих на забезпечення стабільної роботи вебресурсу, безпеки даних та зниження ризиків.

Аудит безпеки вебресурсу є важливою складовою для забезпечення його стійкості до потенційних загроз і вразливостей. Зазвичай, визначають такі критерії оцінки виявлених вразливостей, як:

1. Severity (Серйозність).
2. Responsibility (Відповідальність).
3. Assignment (Призначення).
4. Status (Статус).

Ці критерії грають важливу роль у пріоритизації та управлінні вразливостями. Вони дозволяють оцінити серйозність проблеми, призначити відповідальних осіб чи команди для її вирішення та відслідковувати стан виправлення. Такий підхід допомагає ефективно управляти безпекою та ризиками в вебресурсах.

Аналіз останніх досліджень і публікацій. Поряд з такими етапами проведення аудиту безпеки вебресурсу [1], як планування, що включає визначення мети аудиту та складання плану дій; збір інформації, що передбачає збір загальної інформації про вебресурс, його архітектуру, функціональність, потенційні слабкі місця та оцінка загроз і ризиків, пов'язаних з вебресурсом [2,3]; необхідно провести сканування та виявлення вразливостей, що, в свою чергу, включає:

– Вибір інструментів і методів для аналізу безпеки.

– Використання автоматизованих інструментів для сканування вебресурсу на предмет виявлення потенційних вразливостей.

– Оцінка виявлених вразливостей на серйозність і потенційний вплив на безпеку вебресурсу.

– Підготовка звіту про проведення аудиту безпеки з описом виявлених вразливостей та рекомендацій щодо виправлення.

Для проведення періодичного аудиту безпеки існує безліч інструментів для сканування та виявлення вразливостей, що варіюються за функціоналом, типом вразливостей, які вони можуть знайти, та рівнем

деталізації наданих звітів. Всі ці інструменти сканують вебресурси на предмет різних вразливостей, таких як відкриті порти, використання застарілих версій ПЗ, SQL-ін'єкції, XSS, недоліки аутентифікації, слабкі паролі та інші [4,5].

Мета статті. Провести тестування безпеки для аналізу автоматизованих інструментів сканування вебресурсу на предмет виявлення потенційних вразливостей.

Основний зміст роботи. Аналіз інструментів сканування та виявлення вразливостей для тестування безпеки вебресурсу структурного підрозділу, створеного на базі системи керування вмістом WordPress, проведено наступними сервісами:

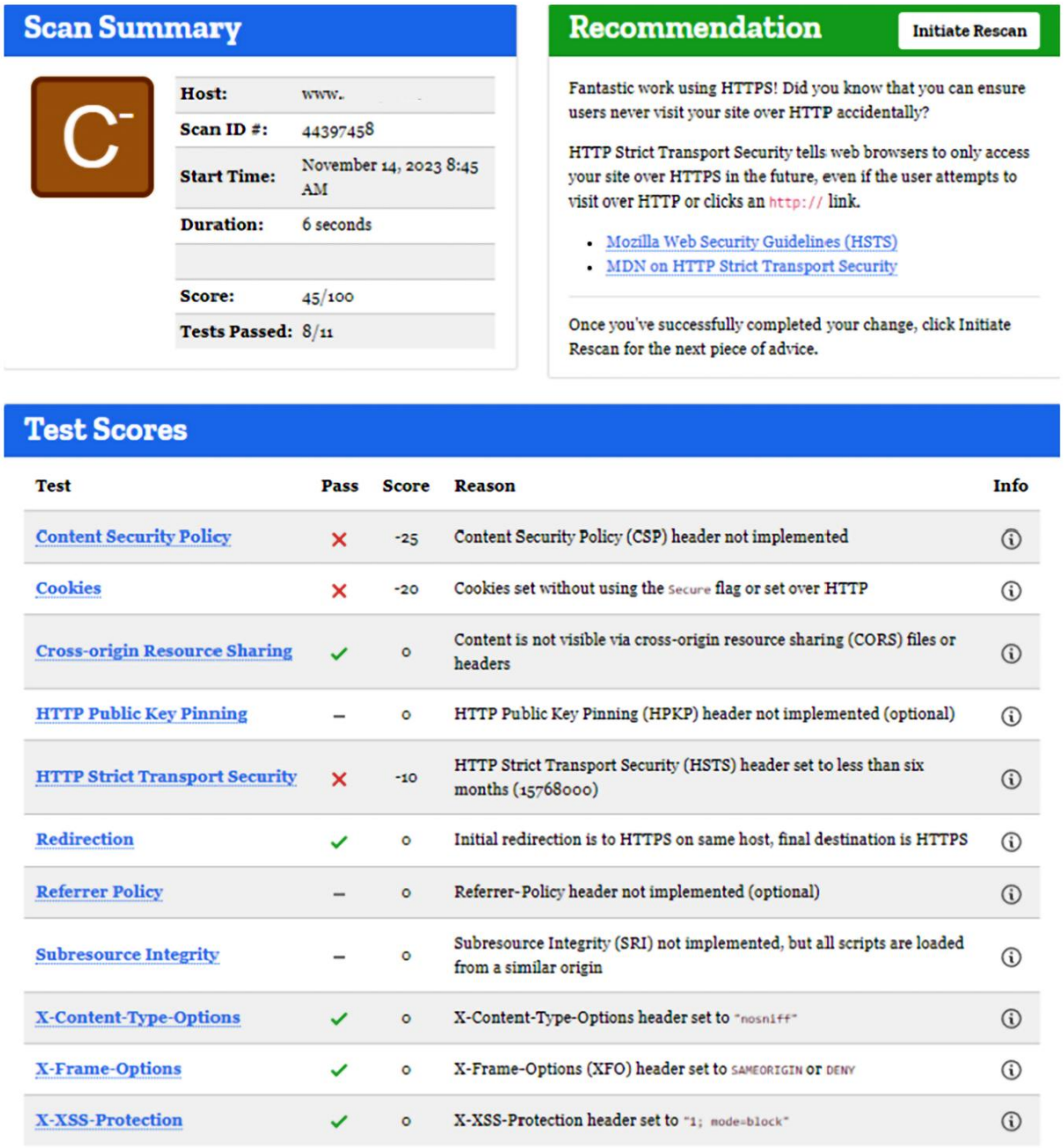
– Mozilla Observatory.

– Qualys.

– ImmuniWeb.

Mozilla Observatory сканує сайти на найпопулярніші вразливості, серед них: потенційно небезпечні cookies, XSS-уразливості та редиректи, також оцінює мережеву безпеку сайтів та роботу механізмів CORS, HPKP, HSTS та інших. Допомагає швидко перевірити технічні параметри і оцінити поточний рівень захищеності сайту.

Проведемо сканування вебресурсу структурного підрозділу (рис.1-2).



Scan Summary

C-

Host:	www.
Scan ID #:	44397458
Start Time:	November 14, 2023 8:45 AM
Duration:	6 seconds
Score:	45/100
Tests Passed:	8/11

Recommendation

Initiate Rescan

Fantastic work using HTTPS! Did you know that you can ensure users never visit your site over HTTP accidentally?

HTTP Strict Transport Security tells web browsers to only access your site over HTTPS in the future, even if the user attempts to visit over HTTP or clicks an `http://` link.

- Mozilla Web Security Guidelines (HSTS)
- MDN on HTTP Strict Transport Security

Once you've successfully completed your change, click Initiate Rescan for the next piece of advice.

Test Scores

Test	Pass	Score	Reason	Info
Content Security Policy	✗	-25	Content Security Policy (CSP) header not implemented	ⓘ
Cookies	✗	-20	Cookies set without using the `Secure` flag or set over HTTP	ⓘ
Cross-origin Resource Sharing	✓	0	Content is not visible via cross-origin resource sharing (CORS) files or headers	ⓘ
HTTP Public Key Pinning	–	0	HTTP Public Key Pinning (HPKP) header not implemented (optional)	ⓘ
HTTP Strict Transport Security	✗	-10	HTTP Strict Transport Security (HSTS) header set to less than six months (15768000)	ⓘ
Redirection	✓	0	Initial redirection is to HTTPS on same host, final destination is HTTPS	ⓘ
Referrer Policy	–	0	Referrer-Policy header not implemented (optional)	ⓘ
Subresource Integrity	–	0	Subresource Integrity (SRI) not implemented, but all scripts are loaded from a similar origin	ⓘ
X-Content-Type-Options	✓	0	X-Content-Type-Options header set to `"nosniff"`	ⓘ
X-Frame-Options	✓	0	X-Frame-Options (XFO) header set to `SAMEORIGIN` or `DENY`	ⓘ
X-XSS-Protection	✓	0	X-XSS-Protection header set to `"1; mode=block"`	ⓘ

Рисунок 1 - Результат сканування сайту завдяки Mozilla Observatory

Mozilla Observatory перевіряє сайт завдяки тестам, представленим в табл. 1.

Таблиця 1 – Набір тестів інструменту Mozilla Observatory

Назва	Значення
Content Security Policy	Стандарт комп'ютерної безпеки, введений для запобігання міжсайтових сценаріїв, клікджекінгу та інших атак шляхом впровадження коду
Cookies	Застосовується для збереження даних на стороні користувача, може захистити користувача від викрадення даних
Cross-origin Resource Sharing	Технологія сучасних браузерів, яка дозволяє надати вебсторінкам доступ до ресурсів іншого домену
HTTP Public Key Pinning	Дозволяє вебсайтам HTTPS протистояти уособленню зловмисників, які використовують неправильно випущені або іншим чином шахрайські цифрові сертифікати
HTTP Strict Transport Security	Політика безпеки дозволяє відразу ж встановлювати безпечне з'єднання замість використання протоколу HTTP
Redirection	Технологія, що використовується у Всесвітньому павутинні для того, щоб вебсторінка була доступна під кількома URL
Referrer Policy	Використовується для аналітики, логування, оптимізації кешу та ін. Однак також може використовуватися для стеження або крадіжки інформації, виконання побічних ефектів, що призводять до витоку чутливих даних користувача і т.п.
Subresource Integrity	Рекомендація W3C для надання способу захисту доставки вебсайту. Зокрема, перевіряє активи, що обслуговуються третьою стороною, наприклад, мережею доставки контенту. Це гарантує, що ці активи не були скомпрометовані у ворожих цілях
X-Content-Type-Options	Заголовок, який підтримується Internet Explorer, Chrome і Firefox 50+, який повідомляє не завантажувати сценарії та таблиці стилів, якщо сервер не вказує правильний тип MIME. Без цього заголовка ці браузери можуть неправильно виявляти файли як сценарії та таблиці стилів, що призводить до атак XSS
X-Frame-Options	X-Frame-Options - це HTTP-заголовок, який дозволяє сайтам контролювати, як сайт може бути розміщений у фреймі iframe. Clickjacking — це практична атака, яка дозволяє зловмисним сайтам обманом змусити користувачів натиснути посилання на сайті
X-XSS-Protection	Використання може створити уразливості XSS на безпечних вебсайтах. Це не слід використовувати, якщо не потрібна підтримка старих вебпереглядачів, які ще не підтримують CSP

Після сканування вебресурсу інструмент видає звіт з рекомендаціями щодо покращення безпеки та посиланнями на корисні матеріали. Оцінка формується за шкалою від 0 до 100, тобто сайт набирає певну кількість балів — йому присвоюється категорія якості “A”, “B”, “C”, “D”, “F”.

За результатами проведеного сканування сайту присвоєна категорія “C-”, оскільки набрано 45 балів зі 100 можливих. Проведено 8 тестів з 11, завдяки яких виявлено слабкі місця вебсайту:

1. Не реалізовано механізм безпеки вебзастосунків, який використовується для скорочення ризиків, пов'язаних з атаками, такими як впровадження скриптів (XSS) та виконання небажаного коду (ін'єкція).
2. Файли cookie встановлюються без використання прапора Secure або через HTTP.
3. Заголовок HTTP Strict Transport Security (HSTS), що використовується для перемикання користувача, який зайшов по HTTP на HTTPS-сервер, встановлено на менше ніж шість місяців.

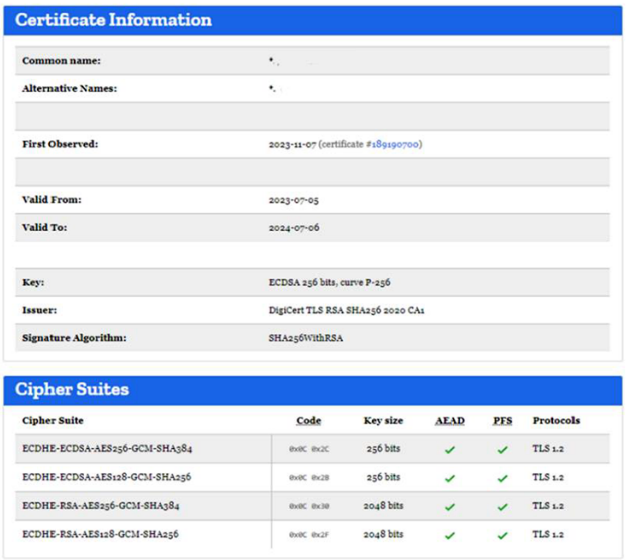


Рисунок 2 - Результат сканування сайту завдяки Mozilla Observatory

При роботі з HTTPS шифрування застосовується у чотирьох випадках: під час обміну ключами, у SSL-сертифікатах, при пересиланні повідомлень та складанні хеш-суми (дайджесту). У кожному з цих випадків використовуються різні набори алгоритмів, про які домовляються клієнт і сервер. Вони вибирають асиметричний шифр для встановлення з'єднання, симетричний шифр для кодування повідомлень та алгоритм хешування для дайджесту.

Для налаштування криптографічних методів, які використовуються сервером, у мережі є спеціальні інструменти. Mozilla пропонує три рекомендовані конфігурації для серверів, які використовують TLS:

1. Сучасна – для роботи з клієнтами, які використовують TLS 1.3 без зворотної сумісності.
2. Проміжна — рекомендована конфігурація більшості серверів.
3. Застаріла - доступ до сервісу здійснюється за допомогою старих клієнтів або бібліотек, таких як IE8, Java 6 або OpenSSL 0.9.8.

За результатами сканування сервіс виявив чотири алгоритми хешування, які належать до протоколу TLS 1.2 таких груп, як:

– SHA384.
– SHA256.

Алгоритми хешування SHA384 і SHA256 належать до стандарту TLS 1.2.

TLS, як і його попередник SSL, — криптографічні протоколи, що забезпечують захищену передачу даних між вузлами у мережі Інтернет. TLS і SSL використовують асиметричне шифрування для автентифікації, симетричне шифрування для конфіденційності та коди автентичності повідомлень для збереження цілісності повідомлень.

Основні функції сертифікату SSL/TLS:

1. Захищати особисті дані.
2. Зміцнювати довіру клієнтів.
3. Відповідати нормативним вимогам.
4. Поліпшити пошукову оптимізацію (SEO).

Наступний інструмент ***Qualys.*** Він проводить сканування вебдодатків та оцінку всіх 10 найбільш критичних ризиків для безпеки вебдодатків OWASP, включаючи міжсайтовий сценарій (XSS), впровадження SQL і розкриття конфіденційних даних.

Інструмент працює наступним чином - спочатку переглядається сертифікат, щоб переконатися, що він дійсний і надійний, а вже потім перевіряється конфігурація SSL у трьох категоріях:

1. Підтримка протоколу.
2. Обмін ключами.
3. Надійність шифру.

Кожна категорія отримує оцінку, ці бали об'єднуються для отримання загальної оцінки 0-100. Нуль у будь-якій категорії призводить до нульової загальної оцінки. Загальна числова оцінка перетворюється на буквену оцінку (A-F). Оцінку A буде підвищено до A+ для винятків конфігурацій і знижено до A-, якщо буде одне або більше попереджень. Інші оцінки, які можна побачити: T (сертифікат не є надійним), M (невідповідність назви сертифіката) і NA (незастосовно, інформацію про сервер SSL не отримано).

Проведемо сканування вебресурсу структурного підрозділу і цим інструментом (рис.3-4).

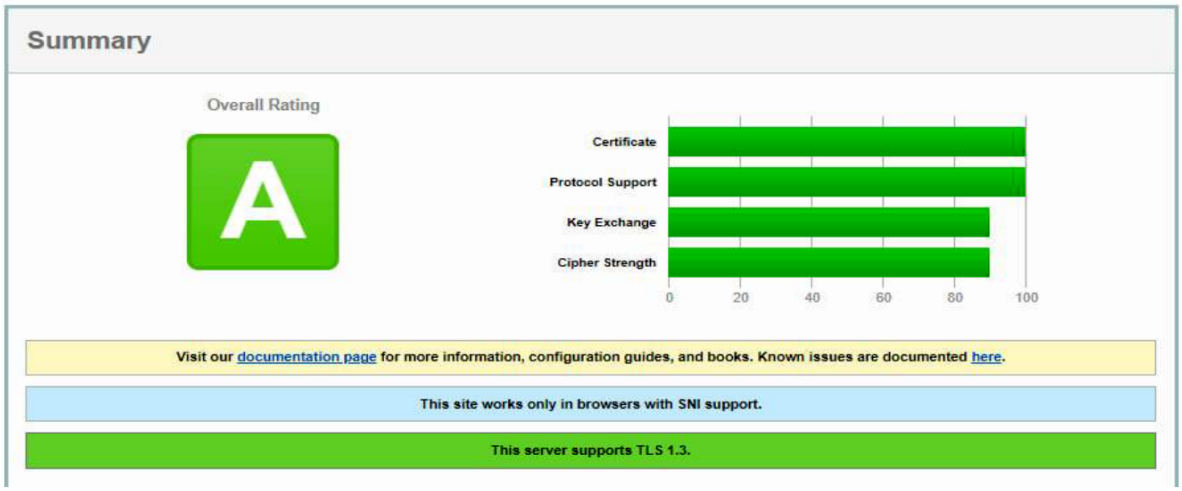


Рисунок 3 - Результат сканування сайту завдяки Qualys

За результатами сканування сайт отримав оцінку "A". Оскільки інструмент виявив TLS протокол декількох версій: TLS 1.2 і TLS 1.3, а також те, що сайт працює тільки в браузерах з підтримкою SNI (Server Name Indication є розширенням протоколу TLS).

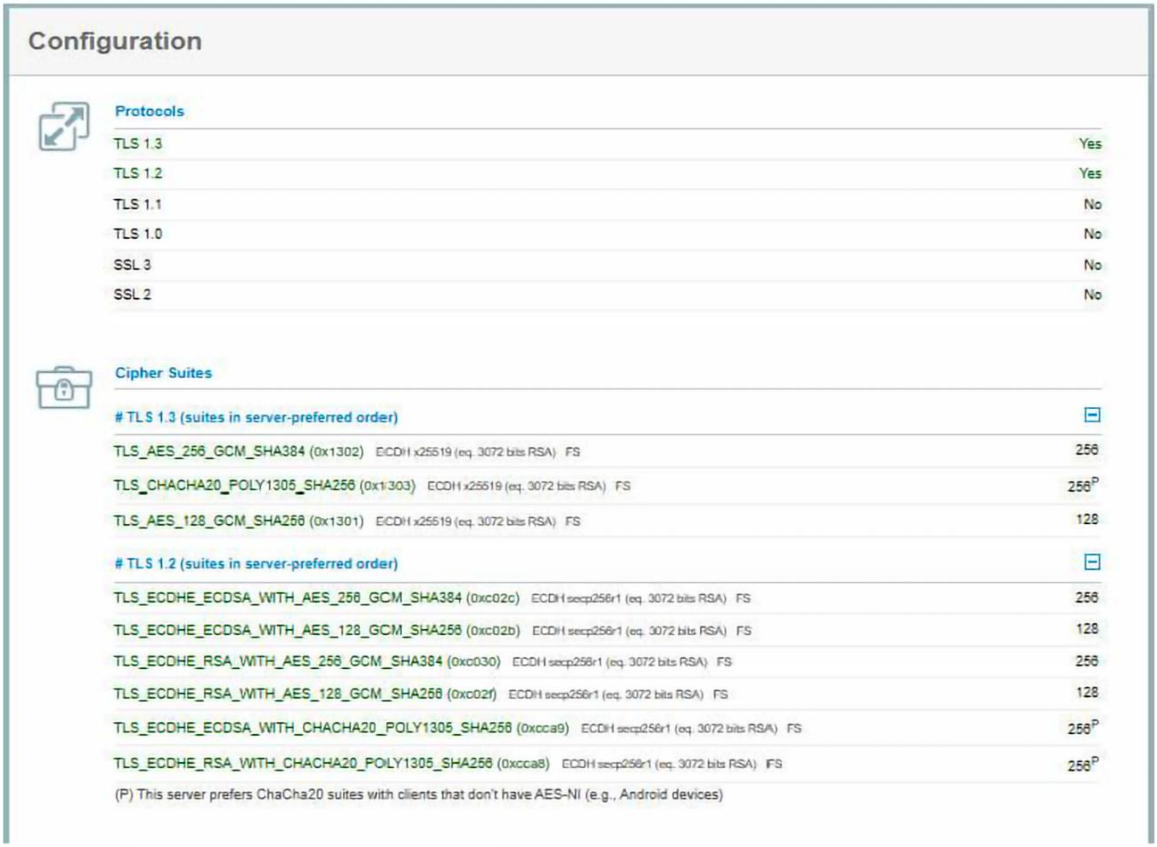


Рисунок 4 - Результат сканування сайту завдяки Qualys

За результатами сканування сервіс виявив три алгоритми хешування, які належать до новішої версії протоколу TLS 1.3:

- SHA384.
- SHA256.
- SHA256 with CHACHA20 POLY1305.

Отже, оцінка "A" виправдана.

І ще один інструмент ***ImmuniWeb***. Є комплексом інструментів для безкоштовної перевірки вебресурсів. Включає такі модулі як: Website Security Test, Cloud Security Test, Email Security Test, Mobile App Security Test, SSL Security Test, Dark Web Exposure Test. Доступний безкоштовний моніторинг безпеки, а також API і CLI-інтерфейс.

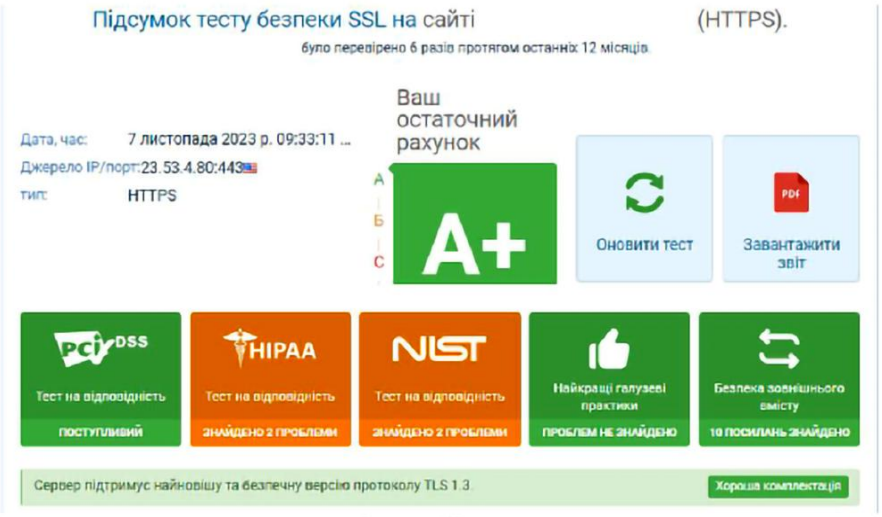


Рисунок 5 - Результат сканування сайту завдяки Immuni Web

За результатами проведеного сканування сайту формується оцінка за шкалою від 0 до 100 і присвоюється категорія якості “A”, “B”, “C”, “F”.

За допомогою даного інструменту вебресурс структурного підрозділу отримав категорію “A+” (рис. 5-б), виявлено, що сервер сайту має хорошу конфігурацію захисту завдяки підтримки сучасних протоколів захисту TLS 1.2 і TLS 1.3.

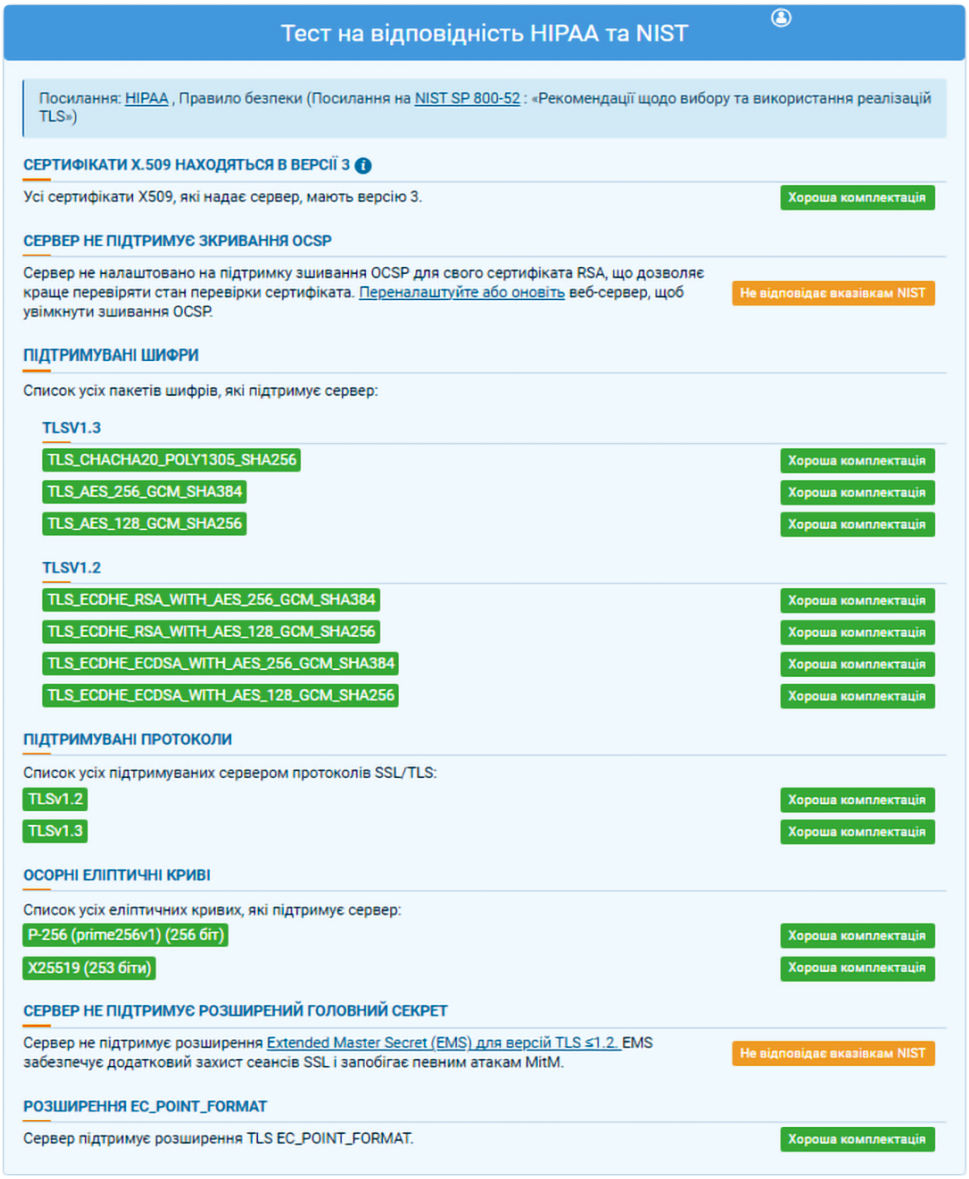


Рисунок 6 - Результат сканування сайту завдяки ImmuniWeb

Також виявлено, що сайт не дотримує усіх вимог щодо протоколів HIPPA та NIST, а саме:

1. Сервер не налаштовано на підтримку зшивання OCSP для свого сертифіката RSA, що дозволяє краще перевіряти стан перевірки сертифіката.
2. Сервер не підтримує розширення Extended Master Secret (EMS) для версій TLS ≤1.2. EMS забезпечує додатковий захист сеансів SSL і запобігає певним атакам MitM.

HIPPA - акт (закон), щоб модернізувати потік медичної інформації, передбачити, як особиста інформація, що зберігається в медичних установах та медичних страхових галузях, має бути захищена від шахрайства та крадіжок.

NIST Cybersecurity Framework – це платформа або керівництво, яка містить набір рекомендацій щодо зниження організаційних ризиків кібербезпеки.

Проведено тестування безпеки вебресурсу структурного підрозділу автоматизованими інструментами сканування на предмет виявлення потенційних вразливостей. В результаті сканування інструментами Qualys та ImmuniWeb, хоча і виявлено недоліки, але все одно присвоєна вища категорія оцінки поточного рівня

захищеності сайту. В той час, як за результатами сканування завдяки інструменту Mozilla Observatory отримана посередня оцінка. Це свідчить про надійніший результат, обумовлений більш широкими можливостями цього сервісу, оскільки він застосовує інтегровані інструменти та рекомендації від OWASP, Probely, а також ті самі Qualys й ImmuniWeb. В цілому вебресурс забезпечує надійний обмін інформацією між браузером користувача та сервером, але завжди є над чим працювати, щоб мати кращий захист.

Так, наприклад, нещодавно були опубліковані нові уразливості, такі як Zombie POODLE, GOLDENDOODLE, 0-Length OpenSSL і Sleeping POODLE, для вебсайтів, які використовують режими блокового шифрування CBC (Cipher Block Chaining). Ці вразливості застосовні, лише якщо сервер використовує TLS 1.2 і версії нижче із режимами шифрування CBC.

У разі оригінального POODLE потрібно вимкнути підтримку SSL 3.0. Однак у цьому випадку є ризик отримати проблеми із сумісністю. Альтернативним рішенням може стати механізм TLS_FALLBACK_SCSV - він гарантує, що обмін даними SSL 3.0 буде проводитися тільки зі старими системами, тим самим зловмисники більше не зможуть ініціювати зниження версії протоколу. Спосіб захисту від Zombie POODLE та GOLDENDOODLE – відключення підтримки CBC у додатках на базі TLS 1.2, але кардинальним рішенням стане перехід на TLS 1.3, так як у новій версії протоколу не використовується CBC-шифрування.

Висновок. Прикладів вразливостей та атак існує величезна кількість. Навіть провівши повний цикл тестування безпеки, не можна бути на 100% впевненим, що вебресурс по-справжньому убезпечено. Але можна бути впевненим у тому, що відсоток несанкціонованих проникнень, крадіжок інформації та втрат даних буде в рази меншим, ніж у тих, хто не проводив тестування безпеки.

Література

1. Vikas V. Web Security Audit and Penetration Testing: Identifying Vulnerabilities and Strengthening Website Security / V. Vikas, G. Saisri, T. Sai Meghana, A. Sree Harshini, G. Kaveri // International Journal for Research in Applied Science & Engineering Technology (IJRASET). – 2023. – Volume 11. Issue VII. – p. 794–805.
2. Sudirman D. Network Penetration dan Security Audit Menggunakan Nmap / D. Sudirman, A. N. Yaqin // SATIN Sains dan Teknologi Informasi. – 2021. – Vol. 7 № 1. – p. 32-44.
3. Zagorodna N. Network Attack Detection Using Machine Learning Methods / N. Zagorodna, M. Stadnyk, B. Lypa, M. Gavrylov, R. Kozak // Challenges to national defence in contemporary geopolitical situation. – 2022. – no. 1. – p. 55-61.
4. Bhanwarlal. XSS and SQL Injection Detection and Prevention Techniques (A Review) / Bhanwarlal, I. Khan // International Journal of Scientific Research in Computer Science, Engineering and Information Technology. – 2022. – Volume 8. Issue I. – p. 53–60.
5. Yesin V. Technique for Searching Data in a Cryptographically Protected SQL Database / V. Yesin, M. Karpinski, M. Yesina, V. Vilihura, R. Kozak, R. Shevchuk // Applied Sciences. – 2023. – Vol.13(20):11525.
6. Галузін І.С. Управління уразливостями корпоративних інформаційних систем на базі рішень QUALYS / І.С. Галузін, Г.Г. Найман // Сучасний захист інформації. – 2021. – № 2(46). – p. 26–31.
7. Dowling B. A cryptographic analysis of the TLS 1.3 handshake protocol. / B. Dowling, M. Fischlin, F. Günther, D. Stebila // J. Cryptol. – 2021. – № 34(4), 37.
8. Chan K.Y. DIDO: data provenance from restricted TLS 1.3 websites / K.Y. Chan, H. Cui, T.H. Yuen // Cryptology ePrint Archive. – 2023. – Paper 2023/1056.

References

1. Vikas V. Web Security Audit and Penetration Testing: Identifying Vulnerabilities and Strengthening Website Security / V. Vikas, G. Saisri, T. Sai Meghana, A. Sree Harshini, G. Kaveri // International Journal for Research in Applied Science & Engineering Technology (IJRASET). – 2023. – Volume 11. Issue VII. – p. 794–805.
2. Sudirman D. Network Penetration dan Security Audit Menggunakan Nmap / D. Sudirman, A. N. Yaqin // SATIN Sains dan Teknologi Informasi. – 2021. – Vol. 7 № 1. – p. 32-44.
3. Zagorodna N. Network Attack Detection Using Machine Learning Methods / N. Zagorodna, M. Stadnyk, B. Lypa, M. Gavrylov, R. Kozak // Challenges to national defence in contemporary geopolitical situation. – 2022. – no. 1. – p. 55-61.
4. Bhanwarlal. XSS and SQL Injection Detection and Prevention Techniques (A Review) / Bhanwarlal, I. Khan // International Journal of Scientific Research in Computer Science, Engineering and Information Technology. – 2022. – Volume 8. Issue I. – p. 53–60.
5. Yesin V. Technique for Searching Data in a Cryptographically Protected SQL Database / V. Yesin, M. Karpinski, M. Yesina, V. Vilihura, R. Kozak, R. Shevchuk // Applied Sciences. – 2023. – Vol.13(20):11525.
6. Galuzin I.S. Vulnerability management of corporate information systems based on QUALYS solutions / I. S. Galuzin, G. G. Naiman // Modern Information Security. – 2021. – № 2(46). – p. 26–31.
7. Dowling B. A cryptographic analysis of the TLS 1.3 handshake protocol. / B. Dowling, M. Fischlin, F. Günther, D. Stebila // J. Cryptol. – 2021. – № 34(4), 37.
8. Chan K.Y. DIDO: data provenance from restricted TLS 1.3 websites / K.Y. Chan, H. Cui, T.H. Yuen // Cryptology ePrint Archive. – 2023. – Paper 2023/1056.

The article discusses the topical issue of testing the security of a web resource of a structural unit created based on the WordPress content management system, using various scanning and vulnerability detection tools, namely Mozilla Observatory, Qualys, ImmuniWeb. Since an important condition for the continuous process of ensuring the security of business processes of a company's website, maintaining business reputation, economic growth and business development are regular diagnostic and recovery procedures of various nature and levels as part of a website security audit aimed at increasing the security and reliability of the Internet resource. Several such works are devoted to searching for vulnerabilities directly in the structure, detecting errors in the code and server software that attackers can use to attack and hack the site. As a result, the security of the structural unit's web resource was tested using the specified automated tools, each of which revealed shortcomings. The result of scanning with the last two tools corresponds to a higher category for assessing the current level of site security. The Mozilla Observatory tool received a mediocre rating, which is due to the broader capabilities of this service. Because it leverages integrated tools and guidance from OWASP, Probely, as well as Qualys and ImmuniWeb, the comprehensive assessment considers more than just securing an encrypted network connection to another system using SSL/TLS. In general, the web resource ensures reliable exchange of information between the user's browser and the server. And with the development of network technologies and the Internet, the interaction of various systems, services, and applications with each other has acquired significant relevance, therefore interaction testing should be approached with the utmost seriousness.
Keywords: *security, web resource, vulnerability, cryptographic algorithms, scanning, certificate, testing.*

Деркач М.В. – к.т.н., доц, доцент кафедри комп'ютерних наук та інженерії Східноукраїнського національного університету імені Володимира Даля, доцент кафедри кібербезпеки Тернопільського національного технічного університету імені Івана Пулюя, e-mail: gln459@gmail.com

Хомишин В.Г. – асистент кафедри кібербезпеки Тернопільського національного технічного університету імені Івана Пулюя, e-mail: homyshyn@gmail.com

Гудзенко В.О. – здобувач вищої освіти Тернопільського національного технічного університету імені Івана Пулюя, e-mail: gfd.lah@gmail.com

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя (Україна)
Університет імені П'єра і Марії Кюрі (Франція)
Маріборский університет (Словения)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Наукове товариство ім. Т.Шевченка

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

ПРОГРАМА

XIII Міжнародної науково-технічної конференції молодих учених та студентів

11-12 грудня 2024 року

УКРАЇНА
ТЕРНОПІЛЬ – 2024

004.056.53
В.Г. Хомишин, аспірант
Тернопільський національний технічний університет імені Івана Пулюя, Україна

ЗАХИСТ ASP.NET CORE ВЕБ-ДОДАТКІВ ВІД ВПРОВАДЖЕННЯ SQL-КОДУ

V.G. Khomyshyn
PROTECTING ASP.NET CORE WEB APPLICATIONS FROM SQL CODE INJECTION

Атаки шляхом впровадження SQL коду є однією з найнебезпечніших загроз для застосування. Зловмисники створюють простий шкідливий код, який вони відправляють у наш додаток у вигляді традиційного введення на основі форм або шляхом налаштування URL-адрес і рядків запиту для виконання довільного коду у нашій базі даних. Ця вразливість може надати доступ зловмисникам до всієї нашої бази даних, тому дуже важливо знаходити та усувати будь-які подібні вразливості у своїх додатках. SQL має архітектурну проблему, характерну для багатьох мов запитів: дані та команди знаходяться в одному рядку. Якщо SQL-запит є результатом поєднання рядків, деякі з яких були введені користувачем, це може бути фактором ризику. Розглянемо два найпоширеніших шляхи вирішення проблеми впровадження SQL коду:

1. Використання підготовлених інструкцій. Впровадження SQL-коду працює, тому що зловмиснику вдається переключити контекст усередині SQL. Варто пам'ятати, що інструкція SQL – це рядок із командами та даними. Екранування введення може бути перспективною стратегією. Коли ми поміщаємо дані користувача в SQL-запит, це також вважається вводом, тому це правило можна застосовувати. Проте виявляється, що правильно екранувати спеціальні символи SQL не так просто, тому що тут дуже багато варіантів залежно від контексту. Усередині рядків на думку приходять одинарні лапки (і заміна їх на \' або ', залежно від бази даних, що використовується).

Щодо інших спеціальних символів: ( ) [ ] – групування, _ % – підстановочні знаки; ; – завершення запиту; -- # /* – коментар, – все залежить від конкретної системи баз даних.

Варіантів кілька, тож тут легко помилитися. Чому б не дозволити комусь іншому зробити всю тяжку роботу? Виявляється, що всі відповідні баз даних для платформи .NET підтримують підготовлені інструкції (які ще називають параметризованими запитами). У них SQL-запит відправляється до бази даних у два етапи. Спочатку створюється інструкція, яка може містити параметри. На другому етапі ці параметри набувають значень. Лише після цього інструкція виконується. Це простий, але ефективний підхід, тому що він нейтралізує проблему SQL архітектури: тепер зрозуміло, де команда, а де дані. Надаючи значення цим заповнювачам, база даних знає, що подана інформація повинна розглядатися лише як дані. Насправді, так ми відключили використання небажаного SQL-коду. Ось як може виглядати запит із параметрами:

```
SELECT * FROM users WHERE email=@email AND password=@password
```

Зверніть увагу, що перед параметрами стоїть символ @, а лапки навколо параметрів відсутні (інакше імена параметрів використовувалися буквально, як рядки). Існують різні варіанти синтаксису для підготовлених інструкцій. Той, що ми використовували досі (з префіксом @), є найбільш поширеним і зручним, оскільки кожен параметр може мати унікальне ім'я, що запам'ятовується. Також для позначення параметрів можна використовувати знак запитання (?), а потім надавати значення за розташуванням, а не по імені. Очевидно, що такий варіант більш схильний до помилок, оскільки розташування параметра буде змінюватися, як тільки десь перед ним буде доданий інший параметр.

Відомий виняток – СУБД Oracle, оскільки імена параметрів тут починаються з двокрапки, і інструкція SQL може виглядати так:

```
SELECT * FROM users WHERE email=:email AND password=:password
```

Протягом тривалого часу найпоширенішим способом надсилання SQL-запитів до бази даних із додатку ASP.NET була технологія ADO.NET (Advanced Data Objects для .NET). Цей підхід також працює з ASP.NET Core. Що стосується Microsoft SQL Server, пакет Microsoft.Data.SqlClient (www.nuget.org/packages/Microsoft.Data.SqlClient) містить постачальника даних. Є аналогічні пакети й інших основних баз даних, але їх способи реалізації підготовлених інструкцій переважно схожі. Якщо ми дійсно працюємо з SQL, то підготовлені інструкції – мабуть, найкращий засіб захисту від впровадження SQL-коду. Однак можна взагалі не писати SQL-код.

2. Використання Entity Framework Core. Щоб позбавити розробників писати SQL-команди, компанія Microsoft пропонує інструмент Entity Framework (EF) Core. Це проста, кросплатформова версія популярної технології доступу до даних, яка дозволяє розробникам .NET працювати з базою даних за допомогою об'єктів .NET та усуває потребу у більшій частині коду для доступу до даних, який, зазвичай, доводиться писати. У EF Core доступ до даних здійснюється за допомогою моделі. Модель складається з класів сутностей та об'єкта контексту, що представляє сеанс взаємодії з базою даних. Об'єкт контексту дозволяє виконувати запити та зберігати дані. Псевдокод, який буде запитувати всіх користувачів з бази даних, щоб знайти тих, хто має певну комбінацію адреси електронної пошти та пароля, має вигляд:

```
db.Users.Where(user => user.email == email && user.password == password)
```

Цей код робить майже те саме, що і SQL з попередніх прикладів, за винятком деяких особливих випадків, таких як бази даних з урахуванням регістру. Однак, тут немає конкатенації строк, а також відсутні параметри. Натомість є об'єкт, що позначає всіх користувачів в історії даних (db.Users) та синтаксис мови LINQ для вибору користувачів, які відповідають певним критеріям. У зловмисника немає можливості впровадити SQL, оскільки його немає. Ну, взагалі він є, але Entity Framework Core генерує його автоматично.

Але насправді це не зовсім повна картина. Entity Framework Core все ж таки дозволяє відправку SQL-запитів безпосередньо в базу даних. Це буде працювати аналогічно фрагменту коду, який ми щойно продемонстрували:

```
var users = db.Users.FromSqlRaw("SELECT * FROM users WHERE
  email='villain@example.com' AND password='noclue'").ToList();
```

Entity Framework Core не перевіряє SQL, він просто перенаправляється до бази даних. Якщо ми використовуємо тут конкатенацію рядків і дані користувача, то програма раптово знову стає вразливою для впровадження SQL-коду. Зручно, що є підтримка параметризованих запитів, де використовується синтаксис, аналогічний синтаксису методу платформи .NET, String.Format(). Але варто переконатись, що ми використовуємо його лише безпосередньо у виклику методу FromSqlRaw. Ця інструкція належним чином екранує значення для параметрів. В якості альтернативи можна використовувати інтерполяцію рядків.

Використання інструментів об'єктно-реляційного відображення, таких як Entity Framework Core, при всій своїй зручності, може створювати деякі проблеми. При їх застосуванні SQL-запити генеруються на основі наданої інформації (наприклад, об'єкта та його зв'язків). У деяких ситуаціях результат згубно впливає на продуктивність. Особливо це може статися, коли в одному запиті об'єднуються кілька таблиць. При роботі з таблицями без первинного ключа SQL може бути більш підходящим способом. Тому навіть найзатятіші прихильники Entity Framework Core завжди шукатимуть шляхи, в яких оригінальні SQL-запити є найкращим варіантом.